Subtyping and Algebraic Data Types

Sven-Olof Nyström

December 18, 2015

Abstract

We present a new method for type inference. Traditional approaches rely on an inductively defined domain of types. Instead, we specify the properties of the type system as a set of axioms, and give a polynomial-time algorithm for checking whether there is any domain of types for which the program types. We show the correctness of the algorithm and also prove that safety properties are satisfied; any program accepted by the type system will not cause type errors at run-time.

The advantages of our approach is that is simpler and more general. The algorithm for checking that a program types is a simple mechanism for propagating type information which should be easy to extend to support other programming language features. The type checking algorithm is more general than other approaches as the algorithm will accept any program that types under any type system that satisfies the axioms.

We also show how to make type information available to compilers and other development tools through an algorithm to determine entailment.

The language we consider is lambda calculus extended with a constructors and a form of type construct we call *open case expressions*. Open case expressions allow a program to manipulate abstract data types where the sets of constructors overlap.

1 Introduction

In the context of functional programming the goal of subtyping has been to generalise Hindley-Milner [11, 16] type inference by allowing one type to include another type. Mitchell described a subtyping system that allowed coercions between types. The coercions might be user-defined or given in the

programming language [17, 18]. 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's subtyping system included a universal type [28]. This would allow a type system to combine the advantages of static and dynamic typing as values where the type was not known could be typed with the universal type. There are several recent attempts to integrate static and dynamic typing that rely on some form of subtyping but not in combination with type inference, for example [27, 26, 33, 6, 13, 36]. These systems rely on run-time type checks in the conversion from dynamically typed values to values with static types. In this context, a generalisation of Hindley Milner type inference to subtyping promises precisely defined safety properties, a reduced need for explicit type declarations and no type tests at run-time.

The traditional approach to show that a program types is to map each program variable to some pre-defined domain of types. However, there are other ways to go. Barendregt et al. [4, Part 2, Chapters 7 and 11] give some examples of type systems expressed as *type algebras*, i.e., a set of types closed under some operations and satisfying a list of axioms. This paper is influenced by that idea, but takes things further by showing that it is possible to give an algorithm that checks whether there is *some* type algebra for which the program types.

We exploit a well-know connection between type inference and logic. As noted by several authors (for example Wand [34], Aiken [2], Palsberg [21, 14]), many type systems allow the problem of typing a given program to be translated to the problem of solving a constraint system. This holds in particular for the subtyping system we consider. Thus, for a given program we can express the typing problem as a set of constraints. This constraint system together with the set of axioms for the type system forms a theory in first order predicate logic.

How do we show that there is some domain of types that satisfies the relevant axioms and types the program without explicitly defining that domain? The approach taken in this paper is to show that the program is *logically consistent* with the axioms for the type system. While the standard approach is to look for a solution to the constraint system, we consider the algebraic properties of types; thus, by checking that the theory is logically consistent we verify that the program types. It follows by Gödel's well-known result ([8], [31, Chapter 3]) that any consistent theory has a model, thus if the theory is logically consistent there is some domain of types that satisfies the constraint system.

Our algorithm for checking consistency is based on simple mechanism for propagating type information. The core algorithm is not new; several

authors describe very similar algorithms [14, 21, 5] for the use in subtyping systems. However, we present a new result; a proof that the algorithm can be used for checking logical consistency.

Thus our algorithm can decide whether there is some type system under which the program types. We show a *subject reduction property*, i.e., that if the program types under any type system that satisfies the axioms then typing is preserved under reduction. This in turn guarantees the usual safety properties, i.e., that a program that types will not cause a type error at run-time.

Since the constraint system is a representation of the types present in the program, it will contain information about program variables, expressions, function parameters and so on. Type information is not explicit, but we show how it can be made available (for the use in, say, compilers or debugging tools) by extending the consistency checking algorithm to deduce additional information of the constraint system (this can be done without an increase in complexity). Using this information, we can determine whether an arbitrary constraint is *entailed* by the constraint system, i.e., whether a constraint can be derived from the constraint system. This allows us to answer questions such as (for example): is the type of a program variable a subtype of some given type, or is the type of one program variable a subtype of another.

The simplicity of our approach has practical advantages; programming language implementations tend to be complicated even when the algorithms they implement are simple, but a more complex algorithm will of course make the implementation even more complex, and a simpler system will be easier to extend.

The language we consider is a generalisation of lambda-calculus which also allows constructors. The language also includes *open case expressions*, a form of case expressions in which those values that do *not* match a given pattern constitute a more narrowly defined subtype. Open case expressions allow a program to manipulate abstract data types where the sets of constructors overlap.

The contents of the rest of this paper is as follows. The programming language is defined in Section 2 and the constraint language in Section 3. The typing rules are given in Section 4 and the proof of the subject reduction property in Section 5. Section 6 presents the algorithm for checking consistency, determines its complexity and shows its correctness. Section 7 presents an algorithm for checking entailment, based on the previous algorithm for checking consistency. Section 9 presents related work, and Section 10 summarises the paper.

2 Language

Our language is a generalisation of lambda-calculus, extended with a set of constructors and a new form of case expressions. A constructor may be applied to a sequence of values to produce a new value (we will refer to a term of this form as a *constructor term*). The language also includes *open case terms*. An open case term takes the form

$$\begin{aligned}&\mathsf{case}(M,\\ &\quad \langle c\ x_1 \ldots x_n\rangle \Rightarrow N,\\ &\quad y \Rightarrow P).\end{aligned}$$

If the term M evaluates to a value that matches the pattern $\langle c\ x_1 \ldots x_n\rangle$, the first branch, the term N, is selected. The second branch is only selected when the pattern does not match. We take advantage of this knowledge when designing the type rules, thus the type of the variable y is a subtype of the of the type of the term M and only allows those possible values of M that do not match the pattern. The type rules given in the next section reflect that the values that the variable y may be bound may be a strict sub-type of the type of the term M. This syntactic form was first considered by Heintze [9] in the context of set-based analysis.

Let $c \in \mathcal{C}$ be a set of constructors. Each constructor has an arity, i.e., a non-negative integer. We will sometimes write $\mathsf{arity}(c)$ to indicate the arity of a constructor c.

Let the set of terms $M, N, P, Q, R \in \mathsf{Term}$ be inductively defined as follows:

1. $x \in \mathsf{Term}$, for any variable x.
2. $\lambda x.M \in \mathsf{Term}$, for $x \in \mathsf{Var}$ and $M \in \mathsf{Term}$.
3. $MN \in \mathsf{Term}$, for M and $N \in \mathsf{Term}$
4. $\langle c\ M_1 \ldots M_n\rangle \in \mathsf{Term}$, for a constructor c of arity n and $M_i \in \mathsf{Term}$, for $i \leq n$.
5. $\mathsf{case}(M, \langle c\ x_1 \ldots x_n\rangle \Rightarrow N, y \Rightarrow P) \in \mathsf{Term}$, for variables $x_1, \ldots, x_n$ and y, terms M, N and P, and a constructor c of arity n. The variables $x_1, \ldots, x_n$ are assumed to be distinct.

We will always assume that a constructor is used with the correct number of arguments, thus we will sometimes omit reference to the arity of the constructor.

2.1 Reduction rules

We assume the usual convention that substitution $M\,[x := N]$ is implemented with renaming so that no free variable in N becomes bound when substituted into M. For further details, see [3].

The relation $\longrightarrow$ over lambda-terms is the least relation which satisfies the following, for arbitrary terms, constructors and variables:

$$(\lambda x.M)N \longrightarrow M\,[x := N]$$

$$\mathsf{case}(\langle c\ M_1 \ldots M_n\rangle, \langle c\ x_1 \ldots x_n\rangle \Rightarrow N, y \Rightarrow P) \longrightarrow N\,[x_1 := M_1, \ldots, x_n := M_n]$$

$$\mathsf{case}(\lambda x.M, \langle c\ \ldots\rangle \Rightarrow N, y \Rightarrow P) \longrightarrow P\,[y := \lambda x.M]$$

$$\mathsf{case}(\langle c\ M_1 \ldots M_n\rangle, \langle c'\ x_1 \ldots x_m\rangle \Rightarrow N, y \Rightarrow P) \longrightarrow P\,[y := \langle c\ M_1 \ldots M_n\rangle]$$

The first reduction rule is the standard rule of lambda calculus. The second rule says that if the first argument of the case term matches the pattern, the corresponding branch is evaluated, with the appropriate substitutions. The third rule assumes that a lambda expression will never match, thus the second branch of the case is evaluated, with the lambda term substituted for y. In the final rule, the constructors c and c' are assumed to be distinct. The first argument is a constructor term that does *not* match the pattern, and the second branch is evaluated.

We define a lambda term to be a *redex* if it can be on the left hand side in one of the rules above, i.e., if it is of one of the forms

1. $(\lambda x.M)N$,
2. $\mathsf{case}(\langle c\ \ldots\rangle, \ldots, \ldots)$, and
3. $\mathsf{case}(\lambda x.M, \ldots, \ldots)$.

Clearly, for any redex M there is some lambda term M' such that $M \longrightarrow M'$. We follow the standard approach in lambda calculus and say that a term M *reduces* to a term M' if M contains a redex N, it holds that $N \longrightarrow N'$, and M' is the result of replacing N with N' in M. Write $M \longrightarrow\!\!\!\!\rightarrow N$ to indicate that M can be reduced to N in a single step and $M \longrightarrow\!\!\!\!\rightarrow^* N$ if M can be reduced to N in zero or more steps.

3 Constraints

Let $X, Y, Z, W \in \mathsf{TVar}$ be the infinite set of *type variables*. Let $c, d, e \in \mathcal{C}$ be the set of constructors. Each constructor has an arity, written $\mathsf{arity}(c)$. For the representation of function types we reserve a constructor c_λ with arity 1 that does not occur in any term.

Let the set of *type expressions* $t, u, v, w \in \mathsf{TExp}$ be defined as follows:

1. $0 \in \mathsf{TExp}$,
2. $\mathsf{TVar} \subseteq \mathsf{TExp}$, and
3. $\langle c\ t_0 \dots t_n \rangle \in \mathsf{TExp}$, if $\mathsf{arity}(c) = n$, and $t_i \in \mathsf{TExp}$, for $i \leq n$.

For type expressions t and u we write $t = u$ to indicate that the two expressions are syntactically equal. Note that a type constructor of arity n takes $n+1$ arguments. As we will see later, the first argument to a type constructor (in the definition, t_0) is contravariant and is needed to allow function types to be represented by type constructors. We will refer to types that are the result of an application of a type constructor as *constructor expressions*. If t is a constructor expression $\langle c\ t_0\ t_1 \dots t_n \rangle$ we will sometimes write $I_i^c(t)$ when referring to the ith element of t, thus $I_i^c(t) = t_i$ for $0 \leq i \leq n$.

Let the set of *constraints* $\varphi \in \mathsf{Constraint}$ be formulas of the following forms (where $\bot$ is the inconsistent constraint):

1. $t \leqslant u$, for $t, u \in \mathsf{TExp}$,
2. $X \restriction S \leqslant t$, for $X \in \mathsf{TVar}$, $t \in \mathsf{TExp}$, and $S \subseteq \mathcal{C}$, and
3. $\bot$

where S is a (finite or infinite) set of constructors.

A filter $X \restriction S$ restricts the type X to the set of constructors indicated by S. Filters are used in the typing of open case terms and in the definition of algebraic data types. We chose to restrict the constraint language to only allow filters in one position in the constraints. This greatly simplifies the derivation rules of the constraint language and various proofs. We will use $t \backslash S$ as a shorthand for $t \restriction (\mathcal{C} \backslash S)$, i.e, if there is a set $S' = \mathcal{C} \backslash S$, then $t \backslash S = t \restriction S'$. For constructors c we will sometimes write $t \restriction c$ for $t \restriction \{c\}$ and $t \backslash c$ for $t \backslash \{c\}$. (Naturally, not all infinite sets S can be represented if one requires constraints to have a finite representation, but we want to allow for the possibility that the constraint language includes *some* constraints with infinite filters. In particular, we assume that for each finite set S, the

complement of S the complement of S has a representation in the constraint language.)

A *constraint system* G is a set of constraints. We will always assume that G is finite.

In the rules below, we assume that Γ is a set $\{x_1 : t_1, \ldots, x_n : t_n\}$ where the x_i are distinct variables, and the t_i are type expressions.

We define a type expression t to be a *subexpression* of another type expression u if either

1. $t = u$, or
2. $u = \langle c\ u_0 \ldots u_n \rangle$ and t is a subexpression of some u_i, for $i \leq \mathsf{arity}(c)$.

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.

In this version of the constraint language filters are only allowed in one particular position in constraints. We had previously considered a more general format in which filters could occur whenever a type expression was expected, but this complicated the constraint solver without offering any additional expressiveness.

It should be noted that we place no restrictions on the constraint systems (other than the syntax given above). Thus we allow, for example, recursive constraints such as

$$\langle c\ X\ Y \rangle \leqslant X.$$

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.

We will refer to a sequence of constraints $t_1 \leqslant t_2, \ldots, t_{n-1} \leqslant t_n$ as a *chain* of constraints, and sometimes write the chain $t_1 \leqslant t_2 \leqslant \ldots \leqslant t_{n-1} \leqslant t_n$. If a constraint system G contains all elements of a chain we will say that G contains the chain.

3.1 Derivation rules for constraints

The derivation rules for constraints given in Figure 1 state that the subtyping relation is reflexive (R) and transitive (T). We also include the standard logic rule that any constraint can be derived from the inconsistent constraint ($\bot$).

General rules

$$\frac{\varphi \in G}{G \vdash \varphi} \quad (\in)$$

$$\frac{}{G \vdash t \leqslant t} \quad (\mathrm{R})$$

$$\frac{G \vdash t \leqslant u, \qquad G \vdash u \leqslant v}{G \vdash t \leqslant v} \quad (\mathrm{T})$$

$$\frac{G \vdash \bot}{G \vdash \varphi} \quad (\bot)$$

Constructor rules

$$\frac{G \vdash t_0 \leqslant u_0, G \vdash t_1 \leqslant u_1, \ldots G \vdash t_n \leqslant u_n}{G \vdash \langle c\ u_0\ t_1 \ldots t_n\rangle \leqslant \langle c\ t_0\ u_1 \ldots u_n\rangle} \quad (\mathrm{C})$$

$$\frac{G \vdash \langle c\ t_0\ t_1 \ldots t_n\rangle \leqslant \langle c\ u_0\ u_1 \ldots u_n\rangle}{u_0 \leqslant t_0} \quad (\mathrm{C}-)$$

$$\frac{G \vdash \langle c\ t_0\ t_1 \ldots t_n\rangle \leqslant \langle c\ u_0\ u_1 \ldots u_n\rangle \qquad 1 \leq i \leq n}{G \vdash t_i \leqslant u_i} \quad (\mathrm{C}+)$$

$$\frac{G \vdash \langle c\ \ldots\rangle \leqslant \langle c'\ \ldots\rangle \qquad c \neq c}{G \vdash \bot} \quad (\mathrm{C}\bot)$$

Filter rule

$$\frac{G \vdash \langle c\ t_0 \ldots t_n\rangle \leqslant X \qquad X \restriction S \leqslant u \qquad c \in S}{G \vdash \langle c\ t_0 \ldots t_n\rangle \leqslant u} \quad (\mathrm{F})$$

Zero rules

$$\frac{}{G \vdash 0 \leqslant t} \quad (\mathrm{Z})$$

$$\frac{G \vdash \langle c\ t_0 \ldots t_n\rangle \leqslant 0}{G \vdash \bot} \quad (\mathrm{Z}\bot)$$

Figure 1: Derivation rules for constraints.

Rule (C) states that any constructor is anti-monotone in its zeroth argument and monotone in its other arguments. Rule (C−) and (C+) state that if two constructor expressions are related, so are their components. A type structure satisfying this property is said to be *invertible* [4]. Two constructor types can only be related if they are applications of the same constructor (C⊥). The filter (F) rule states that applying a filter to a variable will only allow a constructor type to pass if the constructor is a member of the filter. Rule (Z) states that the empty type is a subtype of any other type. Conversely, no constructor type is a subtype of the empty type (Z⊥).

Given a constraint system G and a constraint φ, a derivation $G \vdash \varphi$ is a finite tree where each node is associated with a constraint, the root of the tree is φ and each node is either due to G (in which case it is a leaf) or derived from its children using one of the derivation rules in figures 1. (Note that a constraint derived using one of the rules R or Z is also a leaf.)

We write $G \vdash \varphi$ if there is a derivation of the constraint φ from the constraint system G. If $G \vdash \bot$, we say that G is inconsistent, and, conversely, that G is consistent if $\bot$ cannot be derived. Also note that if $G \subseteq G'$ and $G \vdash \varphi$ then $G' \vdash \varphi$.

3.2 Examples of derivations of constraints

As a simple example of entailment, consider $G = \{X \leqslant Y, Y \leqslant Z\}$. It is of course easy to show $G \vdash X \leqslant Z$ by a single application of Rule (T):

$$\dfrac{\dfrac{(X \leqslant Y) \in G}{G \vdash X \leqslant Y}\,(\in) \qquad \dfrac{(Y \leqslant Z) \in G}{G \vdash Y \leqslant Z}\,(\in)}{G \vdash X \leqslant Z}\,(\mathrm{T})$$

As a slightly more complex example, consider the constraint system

$$G = \{X \leqslant \langle c\ Y\ Z\rangle, Y' \leqslant Y, Z \leqslant Z', \langle c\ Y'\ Z'\rangle \leqslant V\}$$

where c is a constraint of arity 1. It holds that $G \vdash X \leqslant V$. This is one example where it is *not* possible to show entailment without using Rule (C). (Often, entailment can be shown by only using rules that decompose constraints.) The derivation, which is shown in Figure 2, omits all uses of Rule (∈).

4 Typing

The typing rules are given in Figure 3. An environment Γ is a set $\{x_1 : t_1, \ldots, x_n : t_n\}$ where the x_i are distinct variables, and the t_i are type expressions. We will write $\Gamma \Vdash M : t$ to indicate that there is a typing that gives

$$\cdot\ \frac{\dfrac{\dfrac{G \vdash Y' \leqslant Y \qquad G \vdash Z \leqslant Z'}{G \vdash \langle c\ Y\ Z\rangle \leqslant \langle c\ Y'\ Z'\rangle}\,(\mathrm{C})}{G \vdash X \leqslant \langle c\ Y'\ Z'\rangle}\,(\mathrm{T}) \qquad G \vdash \langle c\ Y'\ Z'\rangle \leqslant V}{G \vdash X \leqslant V}\,(\mathrm{T})$$

Figure 2: An example of a derivation of entailment.

the term M the type t in environment Γ. We use the symbol $\Vdash$ for typings to reduce the risk of confusion between derivations in the constraint system and typings.

The first three typing rules are the standard rules of typed lambda calculus [4]. The subsumption rule is fairly standard in subtyping systems [17, 18]. The constructor rule is unremarkable; one unusal feature is perhaps that constructors double as type constructors and value constructors. The rule for case terms is new as it takes into account that in the second branch, only those values that *don't* match the patterns are possible. The rule for typing case could be shortened, if one extended the constraint language to allow filters to be applied to expressions that are not variables. However, the current format of the constraint language allows for simpler proofs in Sections 6 and 7.

As Palsberg and O'Keefe point out [21], Mitchell's subtyping rules allow for any typing an equivalent typing where every other rule is an application of the subsumption rule. As the subtyping relation is reflexive and transitive an instance of the subsumption rule can be inserted anywhere in the typing and two consecutive uses of the subsumption rule can be replaced by one (see also [14] 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. It is easy to see that the same property applies to our system.

It is often convenient to make the constraints of a typing explicit, thus we will sometimes write $\Gamma \Vdash M : t\ [G]$ to indicate that the typing $\Gamma \Vdash M : t$ holds and all constraints in the typing follow from the constraint system G. Naturally, whenever a typing $\Gamma \Vdash M : t$ holds there is always some constraint system G such that $\Gamma \Vdash M : t\ [G]$.

Typing

$$\frac{(x : t) \in \Gamma}{\Gamma \Vdash x : t} \quad \text{(axiom)}$$

$$\frac{\Gamma\,[x \mapsto t] \Vdash M : u}{\Gamma \Vdash \lambda x.M : \langle c_\lambda\ t\ u\rangle} \quad \text{(abstraction)}$$

$$\frac{\Gamma \Vdash M : \langle c_\lambda\ t\ u\rangle \qquad \Gamma \Vdash N : t}{\Gamma \Vdash MN : u} \quad \text{(application)}$$

$$\frac{\Gamma \Vdash M_i : t_i, \quad 1 \leq i \leq n}{\Gamma \Vdash \langle c\ M_1 \ldots M_n\rangle : \langle c\ t_0\ t_1 \ldots t_n\rangle} \quad \text{(constructor)}$$

$$\frac{\begin{array}{c}\Gamma \Vdash M : t \\ t \leqslant X \\ X \restriction c \leqslant \langle c\ u_0 \ldots u_n\rangle \\ \Gamma\,[x_1 \mapsto u_1, \ldots, x_n \mapsto u_n] \Vdash N : w \\ X \setminus c \leqslant Z \\ \Gamma[y \mapsto Z\,] \Vdash P : w\end{array}}{\Gamma \Vdash \mathsf{case}(M, \langle c\ x_1 \ldots x_n\rangle \Rightarrow N, y \Rightarrow P) : w} \quad \text{(case)}$$

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

Figure 3: Subtyping rules for our extension of lambda calculus.

$$\dfrac{\dfrac{\dfrac{[x \mapsto X] \Vdash x : X \qquad X \leqslant Y}{[x \mapsto X] \Vdash x : Y}}{\Vdash \lambda x.x : \langle c_\lambda\ X\ Y \rangle} \qquad \langle c_\lambda\ X\ Y \rangle \leqslant Z}{\Vdash \lambda x.x : Z}$$

Figure 4: The typing of the identity function.

4.1 Examples

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

As a first example, consider the typing of the identity function (Figure 4). The type of the identity function is Z for any X and Y such that $X \leqslant Y$ and $\langle c_\lambda\ X\ Y \rangle \leqslant Z$. It is easy to establish that $\Vdash \lambda x.x : Z$ is satisfied exactly when there is a type W such that $\langle c_\lambda\ W\ W \rangle \leqslant Z$.

The identity function can of course be typed in any type system for lambda calculus. Now consider the lambda term $\lambda f.ff$. The typing of this term will produce a consistent constraint system, as seen in Figure 5. The corresponding constraint system is $\{Y \leqslant \langle c_\lambda\ Z\ X \rangle, Y \leqslant Z, \langle c_\lambda\ W\ W \rangle \leqslant Y\}$. 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 $X = Y = Z = W$, but many subtyping systems in the literature only allow types that can be expressed as finite type expressions and will reject this constraint system.

As a final example, let us consider a lambda-term that will be rejected by our type system. Suppose d and e are distinct constructors of arity zero, and let Γ be an environment containing the bindings $f : \langle c_\lambda\ \langle d\ Z_0 \rangle\ \langle d\ Z_1 \rangle \rangle$ and $x : \langle e\ Z_2 \rangle$. (One 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 6 where the set of encountered constraints is

$$\{\langle c_\lambda\ \langle d\ Z_0 \rangle\ \langle d\ Z_1 \rangle \rangle \leqslant \langle c_\lambda\ X\ Y \rangle, \langle e\ Z_2 \rangle \leqslant X\}.$$

It is easy to see that this constraint system is inconsistent. The inequality $\langle c_\lambda\ \langle d\ Z_0 \rangle \langle d\ Z_1 \rangle \rangle \leqslant \langle c_\lambda\ X\ Y \rangle$ implies that $X \leqslant \langle d\ Z_0 \rangle$ and $\langle d\ Z_1 \rangle \leqslant Y$. From $X \leqslant \langle d\ Z_0 \rangle$ and $\langle e\ Z_2 \rangle \leqslant X$ follows $\langle e\ Z_2 \rangle \leqslant \langle d\ Z_0 \rangle$, which is an inconsistent constraint.

$$\dfrac{\dfrac{\dfrac{\dfrac{[f \mapsto Y] \Vdash f : Y \quad Y \leqslant \langle c_\lambda\ Z\ X\rangle}{[f \mapsto Y] \Vdash f : \langle c_\lambda\ Z\ X\rangle} \quad \dfrac{[f \mapsto Y] \Vdash f : Y \quad Y \leqslant Z}{[f \mapsto Y] \Vdash f : Z}}{[f \mapsto Y] \Vdash ff : X}}{\Vdash \lambda f.ff : \langle c_\lambda\ Y\ X\rangle} \qquad \dfrac{[x \mapsto W] \Vdash x : W \quad \langle c_\lambda\ W\ W\rangle \leqslant Y}{\lambda x.x : Y}}{\Vdash (\lambda f.ff)(\lambda x.x) : X}$$

Figure 5: A lambda-term with a recursive type. The typing of the identity function is simplified.

$$\dfrac{\dfrac{\Gamma \Vdash f : \langle c_\lambda\ \langle d\ Z_0\rangle\ \langle d\ Z_1\rangle\rangle \qquad \langle c_\lambda\ \langle d\ Z_0\rangle\ \langle d\ Z_1\rangle\rangle \leqslant \langle c_\lambda\ X\ Y\rangle}{\Gamma \Vdash f : \langle c_\lambda\ X\ Y\rangle} \qquad \dfrac{\Gamma \Vdash x : \langle e\ Z_2\rangle \qquad \langle e\ Z_2\rangle \leqslant X}{\Gamma \Vdash x : X}}{\Gamma \Vdash fx : Y}$$

Figure 6: A lambda-term where the corresponding constraint system is inconsistent. We assume that d and e are distinct constructors with arity zero and Γ is an environment containing the bindings $f : \langle c_\lambda\ \langle d\ Z_0\rangle\ \langle d\ Z_1\rangle\rangle$ and $x : \langle e\ Z_2\rangle$.

5 The subject reduction property

This section shows that if M is a term such that $M \longrightarrow\!\!\!\!\!\rightarrow N$, and $\Gamma \Vdash M : t$, some environment Γ and type expression t, then $\Gamma \Vdash N : t$. In other words, typings are preserved under reductions. This means that the type system guarantees safety; if a term types, it can only reduce to terms of that type. Since term reduction can be seen as a generalisation of lazy and eager evaluation of functional programming (or of any combination of these), the subject reduction property is independent of evaluation model. The result presented here is a generalisation of the corresponding result of Mitchell [18].

Before turning our attention to the lemma, we give some basic result regarding adding information to environments and the connection between adding bindings to an environment and applying a substitution to a lambda-term. The results are fairly standard; similar results can be found in text-books on typed lambda calculus, for example [4].

For a constraint system G and environments Γ and Γ', say that $G \vdash \Gamma \leqslant \Gamma'$ if $\Gamma = \{x_1 : t_1, \ldots, x_n : t_n\}$, $\Gamma' = \{x_1 : t'_1, \ldots, x_n : t'_n\}$, and $G \vdash t_i \leqslant t'_i$, all $i \leq n$.

Proposition 5.1. *If* $G \vdash \Gamma \leqslant \Gamma'$ *and* $\Gamma' \Vdash M : t\ [G]$, *then* $\Gamma \Vdash M : t\ [G]$.

Proof. Straight-forward, by induction on M. □

Lemma 5.2. *Suppose that* $\Gamma\,[x_1 \mapsto t_1, \ldots, x_n \mapsto t_n] \Vdash N : u\ [G]$ *and* $\Gamma \Vdash M_i : t_i\ [G]$, $i \geq 1$.
Then $\Gamma \Vdash N\,[x_1 := M_1, \ldots, x_n := M_n] : u\ [G]$.

Proof. By induction on N. □

We have now arrived at the proof of the subject reduction property. Generally speaking, a proof of a subject reduction property divides into two parts; first we need to show for any term M that when $M \longrightarrow N$ and M types, N has the same type. Second, we need to do an induction over terms to show that when a sub-term M of a term M' is replaced by a term that M reduces to, the resulting term will type whenever M' types. The first part is a case analysis over the possible forms of a redex. The second part is a somewhat tedious induction over terms.

We will focus on the first part of the proof but only consider the case when M is a redex and an open case expression.

Lemma 5.3. *If* $M \longrightarrow\!\!\!\!\!\rightarrow N$ *and* $\Gamma \Vdash M : t\ [G]$ *then* $\Gamma \Vdash N : t\ [G]$.

Proof. We consider the case when $M = \mathsf{case}(P, \langle c\ x_1 \dots x_n\rangle \Rightarrow Q, y \Rightarrow R)$. From $\Gamma \Vdash M : t\ [G]$ we have, by the type rules, that there are type variables X and Z, and type expressions u, $w_0, \dots, w_n$ such that

$$\Gamma \Vdash P : u \tag{SR1}$$
$$G \vdash U \leqslant X \tag{SR2}$$
$$G \vdash X \restriction c \leqslant \langle c\ w_0\ w_1 \dots w_n\rangle \tag{SR3}$$
$$\Gamma\,[x_1 \mapsto w_1, \dots, x_n \mapsto w_n] \Vdash Q : t \tag{SR4}$$
$$G \vdash X \setminus c \leqslant Z \tag{SR5}$$
$$\Gamma[y \mapsto Z] \Vdash R : t. \tag{SR6}$$

Suppose $P = \langle c\ P_1 \dots P_n\rangle$. By the reduction rules

$$N = Q\,[x_1 := P_1, \dots, x_n := P_n]\,.$$

It follows by SR1, the type rule for constructors and the subsumption rule that there are type expressions $u_0, u_1, \dots, u_n$ such that $\Gamma \Vdash P_i : u_i$ for all $i \geq 1$ and $G \vdash \langle c\ u_0\ u_1 \dots u_n\rangle \leqslant u$. By SR2 and transitivity we have $G \vdash \langle c\ u_0 \dots u_n\rangle \leqslant X$. By Rule F and SR3 we have

$$G \vdash \langle c\ u_0\ u_1 \dots u_n\rangle \leqslant \langle c\ w_0\ w_1 \dots\ w_n\rangle\,.$$

It follows that $G \vdash u_i \leqslant w_i$, for $i \geq 1$. By Proposition 5.1, the typing SR4 and Lemma 5.2 we have

$$\Gamma \Vdash Q\,[x_1 := P_1, \dots, x_n := P_n] : t.$$

Since $N = Q\,[x_1 := P_1, \dots, x_n := P_n]$, the lemma follows.

Suppose $P = \langle c'\ P_1 \dots P_m\rangle$, where c' is a constructor distinct from c. By the definition of reduction,

$$N = R\,[y := \langle c'\ P_1 \dots P_m\rangle]\,.$$

Again, by the type rule for constructors there must be type expressions $u_0, u_1, \dots, u_m$ such that $\Gamma \Vdash P_i : u_i$ all $i \geq 1$ and $G \vdash \langle c'\ u_0\ u_1 \dots u_m\rangle \leqslant u$. By rule F we have $G \vdash \langle c'\ u_0\ u_1 \dots u_m\rangle \leqslant Z$. From SR6 we have $\Gamma\,[y \mapsto \langle c'\ u_0 \dots u_m\rangle] \Vdash R : t$. By Proposition 5.2 we have

$$\Gamma \Vdash R\,[y := \langle c'\ P_1 \dots P_m\rangle] : t.$$

The lemma follows. □

6 How to determine consistency

Given a constraint system G and a constraint φ, a derivation $G \vdash \varphi$ is a finite tree where each node is associated with a constraint, the root of the tree is φ and each node is either due to G (in which case it is a leaf) or derived from its children using one of the derivation rules in Figure 1. (Note that a constraint derived using one of the rules R or Z is also a leaf.)

We write $G \vdash \varphi$ if there is a derivation of the constraint φ from the constraint system G. If $G \vdash \bot$, we say that G is inconsistent, and, conversely, that G is consistent if $\bot$ cannot be derived.

This section defines a construction G^* that uses a restricted set of the derivation rules. For any constraint system G, G^* can always be computed in polynomial time. We will show (Theorem 6.13) that G is consistent iff G^* does not contain any inconsistent constraints. In other words, consistency of G can be determined by computing G^*. The construction of G^* was influenced by a definition in [21].

Definition 6.1. *Given a constraint system G, define $(G)_n$, for $n \geq 0$, to be the smallest sets that satisfy the following:*

1. $(G)_0 = G$.
2. *For all n, $(G)_n \subseteq (G)_{n+1}$.*
3. *If $n > 0$, n even, and $(G)_{n-1}$ contains the constraint $\langle c\ t_0 \dots t_m \rangle \leqslant \langle c\ u_0 \dots u_m \rangle$ then $(G)_n$ contains the constraints $u_0 \leqslant t_0$ and $t_i \leqslant u_i$, for $i \geq 1$.*
4. *Suppose $n > 0$ and n odd. If $(G)_{n-1}$ contains the constraints $t \leqslant X$ and $X \leqslant u$ then $(G)_n$ contains $t \leqslant u$.*

   If $(G)_{n-1}$ contains the constraints $\langle c\ t_0 \dots t_m \rangle \leqslant X$ and $X \restriction S \leqslant u$, where $c \in S$ then $(G)_n$ contains the constraint $\langle c\ t_0 \dots t_m \rangle \leqslant u$.

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

Proposition 6.2. *Let G be some constraint system. It follows that $G^* = (G)_n$, 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 $(G)_i$. □

The complexity of constructing G^* can be determined by an argument similar to one used by Heintze [9]. First, note that G^* only contains type expressions present in G. Thus if the size of G is n, i.e., G contains n expressions, it follows that G^* contains no more than n expressions. Thus 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 $O(n^3)$.

The definition of G^* might seem unnecessarily restrictive as it would not add to the complexity of computing G^* if Item 4 of the definition was generalised to allow arbitrary expressions instead of a variable. However, it turns out that this seemingly straight-forward change would make the proofs in Section 6.1 more complicated, in particular Proposition 6.12 would need to be restated.

Proposition 6.3. *Suppose that G is a constraint system containing constraints $t_i \leqslant t_{i+1}$, for $i \leq m$.*

For some n there are constraints $u_j \leqslant u_{j+1}$ in $(G)_n$ for $j \leq k$, such that $u_1 = t_1$, $u_k = t_m$, $k \leq m$, and none of the type expressions $u_2, \ldots, u_{k-1}$ is a variable.

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

Definition 6.4. *A constraint is* immediately inconsistent *if it is of one of the forms $\bot$, $\langle c\ \ldots\rangle \leqslant 0$, or $\langle c\ \ldots\rangle \leqslant \langle c'\ \ldots\rangle$, where $c \neq c'$.*

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

Note that a constraint system consisting of a single immediately inconsistent constraint is always inconsistent, but there are constraints such that any constraint system that contains them is inconsistent, yet they are not immediately inconsistent. Consider for example the constraint system

$$G = \{\langle c\ \langle d\ Y\rangle\ \langle e\ Z\rangle\rangle) \leqslant \langle c\ X\ X\rangle)\}$$

where c is a constructor with arity 1 and constructors d and e are distinct constructors with arity 0.

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 time $O(n^3)$, where n is the number of type expressions in the constraint system.

We show in Theorem 6.13 that a constraint system is locally consistent exactly when it is consistent.

Proposition 6.5. *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. It follows that there is some immediately inconsistent constraint in G^*, thus G is not locally consistent.*

Proof. By Proposition 6.3 we can assume that the type expressions $t_2, \ldots, t_{n-1}$ are not type variables, thus none of the type expressions $t_1, \ldots, t_n$ is a type variable.

Suppose that all constraints $t_i \leqslant t_{i+1}$ are *not* immediately inconsistent, for $i \leqslant n$. If the constraint $t_1 \leqslant t_n$ is immediately inconsistent neither t_1 nor t_n is a type variable, thus t_1 is of the form $\langle c \ \ldots \rangle$. It follows that all t_i must be of the same form, thus 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 6.6. *For a constraint system G and a constraint φ write $G \vdash_{RZTC} \varphi$ if φ can be derived from G via a single use of one of the rules R, Z, 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 $i < n$, $G_i^* \vdash_{RZTC} \psi_i$,*
2. *for all $i < n$, $G_{i+1} = G_i \cup \{\psi_i\}$, and*
3. *$\varphi \in G_n^*$.*

Proposition 6.7. *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 give the construction of the constraints systems $H_1, \ldots, H_k$. Showing 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 6.8. *Suppose that G is a constraint system, and that there is a derivation $G \vdash \varphi$ that does not mention $\bot$. 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 6.7 there are constraint systems $G_1, \ldots, G_k$ 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 C− or C+ it follows immediately that $\varphi \in G_k^*$ and thus $G_1, \ldots, G_k \Rightarrow \varphi$.

If the last rule is R, Z, C, or T we have $G_k^* \vdash_{\text{RZTC}} \varphi$ 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 $\bot$, C$\bot$ or Z$\bot$ because of the assumption that the derivation does not mention $\bot$. □

To summarise: for any derivation $G_1 \vdash \varphi$ (that does not involve $\bot$) 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.

6.1 Proof that local consistency is equivalent to consistency

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

Proposition 6.9 (Rule R)**.** *Let G be 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 $n > 0$, the proposition holds for $n - 1$ and $(H)_n$ but not $(H)_{n-1}$ contains the constraint $u \leqslant u'$.

If n is even $(H)_{n-1}$ contains some constraint $\langle c\ w_0 \dots w_m\rangle \leqslant \langle c\ w'_0 \dots w'_m\rangle$ where either $u = w'_0$ and $u' = w_0$, or $u = w_i$ and $u' = w'_i$, some $i \geq 0$. We will only consider the second case as the two cases are nearly symmetric. By the induction hypothesis either $(G)_{n-1}$ contains $\langle c\ w_0 \dots w_m\rangle \leqslant \langle c\ w'_0 \dots w'_m\rangle$ or $\langle c\ w_0 \dots w_m\rangle = \langle c\ w'_0 \dots w'_m\rangle$ and $\langle c\ w_0 \dots w_m\rangle$ is a subexpression of t. It is easy to complete the proof in either case.

If n is odd there are two cases in the construction of $(H)_n$. Suppose $(H)_{n-1}$ contains the constraints $u \leqslant X$ and $X \leqslant u'$, for some type variable X. By the induction hypothesis there are three alternatives. If $(G)_{n-1}$ contains both $u \leqslant X$ 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 alternative, with $X = u'$, is similar. Consider the second case. Suppose u is some constructor expression $\langle c\ \dots\rangle$ and $(H)_{n-1}$ contains $u \leqslant X$ and $X \restriction S \leqslant u'$, some type variable X and set S containing c. By the induction hypothesis (and since $X \restriction S$ cannot be a subexpression of t) $(G)_{n-1}$ contains $X \restriction S \leqslant u'$. If $(G)_{n-1}$ also contains $u \leqslant X$ the proposition follows immediately. We know that $u \neq X$ since u is a constructor expression. □

Proposition 6.10 (Rule Z)**.** *Suppose that G is a constraint system and t some type expression and. Let $H = G \cup \{0 \leqslant t\}$.*

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

1. *$\varphi \in (G)_n$, or*
2. *$\varphi = (0 \leqslant u)$, some type expression u.*

Proof. As usual, by induction on n. The case $n = 0$ is straight-forward. If $n > 0$ and n is even, the proposition follows immediately.

Suppose $n > 0$, n is odd and $(H)_n$ contains $v \leqslant w$ but $(H)_{n-1}$ does not. Note that if $v = 0$ the proposition follows immediately. We consider the case when $v \neq 0$. We have two possible situations. If there is some type

variable X such that $(H)_{n-1}$ contains $v \leqslant X$ and $X \leqslant w$ we have by the induction hypothesis that $(G)_{n-1}$ contains them and the proposition follows. If v is some constructor expression $\langle c \ \ldots \rangle$ and $(H)_{n-1}$ contains $u \leqslant X$ and $X \restriction S \leqslant w$ the induction hypothesis again gives us that $(G)_{n-1}$ must contain the two constrains and thus $(v \leqslant w) \in (G)_n$. □

Proposition 6.11 (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 $w_1 = u$, $w_m = v$. For any $i < m$, the constraint $w_i \leqslant w_{i+1}$ occurs in $(G)_k$.

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 the induction hypothesis gives us that $(H)_{n-1}$ contains constraints $w \leqslant w'$ where w and w' are constructor expressions with some constructor c as constructor, and either $u = I_0^c(w')$ and $u = I_0^c(w)$, or $u = I_i^c(w)$ and $u = I_i^c(w')$, some $i > 0$. We will only consider the second case as the two cases are nearly symmetric. By the induction hypothesis there for some k a chain of constraints $w_1 \leqslant \ldots \leqslant w_m$ in $(G)_k$ such that $w_1 = w$ and $w_m = w'$. By Proposition 6.3 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 constructor expressions with c as constructors, thus $I_i^c(w_j)$ is defined for all $j \leq m$. By the construction of (G) it follows that $(G)_{k+2}$ contains $I_i^c(w_j) \leqslant I_i^c(w_{j+1})$, for $j \leq m$.

If n is odd there are two cases. Suppose we have 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. Consider the second case where $(H)_{n-1}$ contains the constraints $u \leqslant X$ and $X \restriction S \leqslant u'$ and u is some constructor expression $\langle c \ \ldots \rangle$ such that S contains c. In this case there is by the induction hypothesis sequences of type expressions $u_1, \ldots, u_m$ and $v_1, \ldots, v_l$ such that $u_1 = u$, $u_m = X$, $v_l = u'$ and $(G)_{n-1}$ contains the constraints $u_i \leqslant u_{i+1}$, $X \restriction S \leqslant v_1$ and $v_j \leqslant v_{j+1}$, for $i < m$ and $j < l$. By Proposition 6.3 and the assumption that G is locally consistent we can assume that u_{m-1} is of the form $\langle c \ \ldots \rangle$. Since $(G)_{n-1}$ contains $u_{m-1} \leqslant X$ and $X \restriction S \leqslant v_1$ we have $(u_{m-1} \leqslant v_1$ in $(G)_n$. It follows that $(G)_n$ contains the chain $u_1 \leqslant \ldots \leqslant u_{m-1} \leqslant v_1 \ldots \leqslant v_l$. □

Proposition 6.12 (Rule C)**.** *Let G be a constraint system containing the constraints $t_i \leqslant u_i$, for $i \leq n$. Let c be a constructor of arity n and $\varphi = \langle c\ u_0\ t_1 \dots t_n\rangle \leqslant \langle c\ t_0\ u_1 \dots u_n\rangle$ and $H = G \cup \{\varphi\}$. It follows that whenever a constraint ψ occurs in $(H)_n$, we have either*

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

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

Suppose that $n > 0$ and the constraint $v \leqslant v'$ occurs in $(H)_n$ but not in $(H)_{n-1}$. If n is even, $(H)_{n-1}$ contains a constraint $w \leqslant w'$ where w and w' are constructor expressions with some constructor c as constructor, and either $v = I_0^c(w')$ and $v' = I_0^c(w)$, or $v = I_i^c(w)$ and $v' = I_i^c(w')$, some $i > 0$. Consider the second case (the first case is similar). By the induction hypothesis either $(G)_{n-1}$ contains $w \leqslant w'$ or $(w \leqslant w') = \varphi$. In either case the proposition follows immediately.

If n is odd and there is a type variable X such that the constraints $v \leqslant X$ and $X \leqslant w$ occur in $(H)_{n-1}$ it follows by the induction hypothesis that the two constraints also occur in $(G)_{n-1}$, and by the construction of (G), we have $v \leqslant w'$ in $(G)_n$. If $w = \langle d\ \dots\rangle$ and $(H)_{n-1}$ contains $u \leqslant X$ and $X \restriction S \leqslant w$, where $d \in S$, the induction hypothesis implies that both constraints are in $(G)_{n-1}$. □

Theorem 6.13. *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^*. When G is inconsistent it follows by the derivation rules for constraints that there is some immediately inconsistent constraint $t \leqslant u$ such that $G \vdash t \leqslant u$, and by Lemma 6.8 constraint systems $G_1, \dots, G_n$ such that $G_1 = G$ and $G_1, \dots, G_n \Rightarrow t \leqslant u$.

We will show by induction on n that whenever $G_1, \dots, G_n \Rightarrow t \leqslant u$, 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 RZTC-rule is R or Z it follows by Propositions 6.9 and 6.10 that $t \leqslant u \in G^*_{n-1}$ and the induction hypothesis applies.

If the last RZTC-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 6.11, 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 6.5 that there is an immediately inconsistent constraint in G^*, thus G_{n-1} is not locally consistent.

If the last RZTC-rule is C there is some constructor c with arity n and type expressions v_i and w_i, for $i \leq n$, such that G^*_{n-1} contains $v_i \leqslant w_i$, for $i \leq n$, and $G_n = G_{n-1} \cup \{\langle c\ w_0\ v_1 \ldots v_n\rangle \leqslant \langle c\ v_0\ w_0 \ldots w_n\rangle\}$ Suppose G^*_n contains an immediately inconsistent constraint $t \leqslant u$. Since $t \leqslant u$ is immediately inconsistent, it follows that t and u must be constructor expressions using different constructors. By Proposition 6.12 this implies that $(t \leqslant u) \in G^*_{n-1}$ and the theorem follows by the induction hypothesis. □

7 Entailment

In this section we present an algorithm for determining whether a constraint φ is entailed (i.e., whether it can be derived from the constraint system).

Say that a constraint system G is in *normal form* if every constraint in G is of the form $t \leqslant t'$ or $X \restriction S \leqslant t$ and each type expression is either a variable or a constructor expression of the form $\langle c\ X_0 \ldots X_n\rangle$. Any constraint system can be converted to normal form by introducing extra variables. Also note that when G is in normal form so is G^*.

Lemma 7.1. *Suppose that G is a consistent constraint system in normal form and c is a constructor of arity n.*

1. *If $G \vdash X \leqslant \langle c\ t_0 \ldots t_n\rangle$, there are type variables $u_0, \ldots, u_n$ such that $G \vdash u_0 \leqslant t_0$ and $G \vdash t_i \leqslant u_i$, for $i \geq 1$, and $X \leqslant \langle c\ u_0 \ldots u_n\rangle$ in G^*.*

2. *If $G \vdash \langle c\ t_0 \ldots t_n\rangle \leqslant X$, there are type variables $u_0, \ldots, u_n$ such that $G \vdash t_0 \leqslant u_0$ and $G \vdash u_i \leqslant t_i$, for $i \geq 1$, and $\langle c\ u_0 \ldots u_n\rangle \leqslant X$ in G^*.*

Proof. We will only consider the first part of the lemma. By Lemma 6.8 there are constraint systems $G_1, \ldots, G_m$ such that $G_1 = G$ and

$$G_1, \ldots, G_m \Rightarrow X \leqslant \langle c\ t_0 \ldots t_n\rangle\,.$$

The proof is by induction on m. If $m = 1$ we have immediately that $X \leqslant \langle c\ t_0 \ldots t_n\rangle$ in G^*, and all t_i are type variables as G is in normal form.

Suppose that $n > 1$. If the last RZTC-rule is R or Z, the lemma follows by Proposition 6.9 or 6.10 and the induction hypothesis.

Suppose the last RZTC-rule is T. There are by Proposition 6.11 type expressions $v_1, \ldots, v_l$ such that G^*_{n-1} contains $v_i \leqslant v_{i+1}$, all $i < l$, where

$v_1 = X$ and $v_l = \langle c\ t_0 \dots t_l \rangle$. By Proposition 6.3 we can assume that v_2 is not a variable. Since G is assumed to be consistent it follows that v_2 is also a constructor expression with c as constructor. Since $G \vdash v_2 \leqslant v_l$ it also holds that $G \vdash I_o^c(v_l) \leqslant I_0^c(v_2)$ and $G \vdash I_i^c(v_2) \leqslant I_i^c(v_l)$, for $i \geq 1$.

Suppose the last RZTC-rule is C. The lemma follows by Proposition 6.12 and the induction hypothesis. □

Definition 7.2. *Given a constraint system G define $G^\diamond$ as the smallest set of constraints such that*

1. $G \subseteq G^\diamond$,
2. *if $G^\diamond$ contains $\langle c\ t_0\ t_1 \dots t_n \rangle \leqslant \langle c\ u_0\ u_1 \dots u_n \rangle$ then $G^\diamond$ also contains $u_0 \leqslant t_0$, and $t_i \leqslant u_i$, for $i \geq 1$,*
3. *if $G^\diamond$ contains $t \leqslant u$ and $u \leqslant v$ then $G^\diamond$ also contains $t \leqslant v$,*
4. *if $G^\diamond$ contains the constraints $\langle c\ t_0 \dots t_m \rangle \leqslant X$ and $X \restriction S \leqslant u$, where $c \in S$ then $G^\diamond$ also contains the constraint $\langle c\ t_0 \dots t_m \rangle \leqslant u$, and*
5. *if $G^\diamond$ contains the constraints $t \leqslant \langle c\ u_0\ u_1 \dots u_n \rangle$, $\langle c\ v_0\ v_1 \dots v_n \rangle \leqslant w$, $v_0 \leqslant u_0$ and $u_i \leqslant v_i$, for $i \geq n$, then $G^\diamond$ also contains the constraint $t \leqslant w$.*

Note that for any constraint system G, $G \subseteq G^\diamond = (G^\diamond)^\diamond$. It is also easy to see that $G^* \subseteq G^\diamond$. Since the computation of $G^\diamond$ does not introduce any new type expressions, $G^\diamond$ contains at most $O(n^2)$ constraints, where the number of expressions in G is n. Also, if G is in normal form, so is $G^\diamond$.

Lemma 7.3. *Let G be a consistent constraint system in normal form. Suppose there are type variables X, Y such that $G_1, \dots, G_n \Rightarrow X \leqslant Y$, with $G_1 = G$. It follows that $(X \leqslant Y) \in G^\diamond$ or $X = Y$.*

Proof. By induction on n. Suppose $n = 1$. As $G^* \subseteq G^\diamond$ and $(X \leqslant Y) \in G^*$ we have immediately that $(X \leqslant Y) \in G^\diamond$.

Suppose that $n > 1$ and the lemma holds for $n - 1$. If the last rule is R or Z the lemma follows immediately as $X = Y$ or $(X \leqslant Y) \in G_{n-1}^*$.

Suppose the last rule is T. By Proposition 6.11 there is a chain $X \leqslant t_1 \leqslant t_2 \leqslant \dots \leqslant t_m \leqslant Y$ in G_{n-1}^*. By Proposition 6.3 we can assume that there is no variable among the type expressions $t_2, \dots, t_{m-1}$. Since G is consistent each of the expressions must be a constructor expression using the same constructor, thus $t_i = \langle c\ u_0^i \dots u_m^i \rangle$, for $i \geq 1$. By the construction of G_{n-1}^* it follows that G_{n-1}^* contains $u_0^{i+1} \leqslant u_0^i$ and $u_k^i \leqslant u_k^{i+1}$, for $i \geq 0$ and $k \geq 1$. Since G is in normal form all type expressions u_k^i must be variables. Thus,

we can apply the induction hypothesis and find that $G^\diamond$ contains constraints $u_0^{i+1} \leqslant u_0^i$ and $u_k^i \leqslant u_k^{i+1}$, for $k \geq 0$. Since $G^\diamond$ is closed under transitivity we find that $G^\diamond$ contains $u_0^m \leqslant u_0^1$ and $u_k^1 \leqslant u_k^m$. By Item 5 of Definition 7.2 we can conclude that $G^\diamond$ contains $X \leqslant Y$.

Suppose the last rule is C. We have by Proposition 6.12 that $(X \leqslant Y) \in G^*_{n-1}$, and by the induction hypothesis $(X \leqslant Y) \in G^\diamond$. □

To compute $G^\diamond$ efficiently we need to make some assumptions on G. Say that a constraint system G is in *constructor normal form* if it is in normal form and there are sets of variables $\{E_i^k, \ldots\}$ and $\{C^k, \ldots\}$ such that every constructor expression is of the form $\langle c_k\ E_0^k \ldots E_n^k \rangle$ for some k (and $\mathsf{arity}(c_k) = n$), and G contains constraints $\langle c_k\ E_0^k \ldots E_n^k \rangle \leqslant C^k$ and $C^k \leqslant \langle c_k\ E_0^k \ldots E_n^k \rangle \leqslant C^k$, for all k. It should be clear that given a constraint system G an equivalent constraint system G' in constructor normal form can be computed in time proportional to the size of G, and that when G is in constructor normal form, so is $G^\diamond$.

It turns out that the asymptotic complexity of computing $G^\diamond$ is $O(n^3)$, where n is the number of expressions in G. This is the same complexity as was required for computing G^*. It is easy to see that $G^\diamond$ contains at most n^2 constraints. Item 5 of Definition 7.2 is difficult to implement efficiently in general. However, if we assume that G is in constructor normal form we find that Item 5 of Definition 7.2 can expressed as follows:

5 If $G^\diamond$ contains inequalities $E_0^l \leqslant E_0^k$ and $E_i^k \leqslant E_i^l$, for $i \geq 1$, then $G^\diamond$ contains $C_k \leqslant C_l$.

Clearly, this relationship can be maintained by checking $G^\diamond$ for the presence of all constraints of the forms $E_0^l \leqslant E_0^k$ and $E_i^k \leqslant E_i^l$, for $i \geq 1$, whenever one constraint of this form is added. If the maximum arity of constructors is bounded, this check can be done in constant time, if not, an efficient implementation is to maintain for each constraint $E_0^l \leqslant E_0^k$ in $G^\diamond$ a counter indicating the largest value of i for which the constraint $E_i^k \leqslant E_i^l$ is present in the constraint system. If checking for the presence of a particular constraint can be done in constant time, the cost of maintaining Item 5 is proportional to the number of constraints in $G^\diamond$, thus it will not affect the overall complexity of $G^\diamond$. Implementing transitivity (Item 3) requires an $O(n)$ operation whenever a constraint is added, same as for G^*, thus the cost of computing $G^\diamond$ is $O(n^3)$.

[say a few words to explain the construction.]

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 a constructor type $\langle c\ t_{j_0}\ t_{j_1} \ldots t_{j_n} \rangle$, let*

$$\begin{aligned} L_i = \{Z \mid Z \leqslant \langle c\ X_0\ X_1 \ldots X_n \rangle \in G^*, \\ X_0 \in U_{j_0}, X_{j_k} \in L_{j_k}, \textit{for } k \geq 1\}, \\ U_i = \{W \mid \langle c\ X_0\ X_1 \ldots X_n \rangle \leqslant W \in G^*, \\ X_0 \in L_{j_0}, X_{j_k} \in U_{j_k}, \textit{for } k \geq 1\}, \end{aligned}$$

2. *If t_i is a variable let*

$$\begin{aligned} L_i = \{Y \mid (Y \leqslant t_i) \in G^\diamond\}, \\ U_i = \{Y \mid (t_i \leqslant Y) \in G^\diamond\}. \end{aligned}$$

Theorem 7.5. *Suppose that G is a consistent constraint system in normal form, t a type expression with the sequence of subexpressions $\{t_1, \ldots, t_n\}$ ordered so that any expression occurs after its proper subexpressions, and for $i \leq n$ the sets L_i and U_i are defined as above.*

It follows that for any type variable X,

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 is a variable the theorem follows from Lemma 7.3.

If t_i is a constructor type $\langle c\ t_{i_0}\ t_{i_1} \ldots t_{i_m} \rangle$ the sets L_{i_k}, U_{i_k}, for $k \geq 0$, have are already been computed. Suppose that there is some type variable Y such that $G \vdash Y \leqslant t_i$. By Lemma 7.1 there are type expressions $u_0, \ldots, u_m$ such that G^* contains $Y \leqslant \langle x\ u_0 \ldots u_m \rangle$ and $G \vdash u_0 \leqslant t_{i_0}$ and $G \vdash t_{i_k} \leqslant u_k$, for $k \geq 0$. Since G is in normal form G^* is also in normal form. This implies that $u_0, \ldots, u_m$ are variables. By the induction hypothesis we can determine $G \vdash t_{i_0} \leqslant u_0$ and $G \vdash u_k \leqslant t_{i_k}$ by examining the sets U_{i_0} and L_{i_k}, for $k \geq 0$. If (the variables) u_0 and u_{i_k}, for $k \geq 0$ are present in these sets and G^* contains $\langle x\ u_0 \ldots u_m \rangle$ we have $Y \in L_i$. □

The complexity of computing the sets L_i and U_i can be determined as follows. Suppose that L_j and U_j has already been computed, for $j < i$, Assuming that the size of G is n and the number of subexpressions of the type expression t is m, the cost of computing L_i and U_i is proportional to the sizes of G^* and $G^\diamond$ which are $O(n^2)$. Thus the total worst-case cost of

determining whether a constraint $X \leqslant t$ or $t \leqslant X$ is entailed 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 $O(n^3)$.

Given Theorem 7.5 it is straight-forward to determine entailment. If a constraint system G is inconsistent, any constraint can be derived from G. Suppose G is consistent. $G \vdash \varphi$ will only hold if φ is an inequality $t \leqslant u$. If either t or u is a variable, entailment follows directly from Theorem 7.5. If $t \leqslant u$ is immediately inconsistent, it is easy to see that $G \vdash t \leqslant u$ cannot hold. The remaining case is when both t and u are constructor expressions using the same constructor. If t and u are function types $\langle c\ t_0 \ldots t_n \rangle$ and $\langle c\ u_0 \ldots u_n \rangle$, respectively, entailment can be determined recursively, as $G \vdash t \leqslant u$ holds if and only if $G \vdash u_0 \leqslant t_i$ and $G \vdash t_i \leqslant u_u$ hold, for $i \geq 1$.

8 Algebraic Data Types

In this section we will investigate how the constructor concept defined earlier in this paper can be used to define a generalisation of the recursive data type concept found in functional programming languages such as SML and Haskell.

We will consider data type definitions of the form

$$\begin{aligned} \mathsf{data}\ x = &\ \langle c_1\ t_{11} \ldots t_{1n_1} \rangle \\ \cup &\ \langle c_2\ t_{21} \ldots t_{2n_2} \rangle \\ &\ \ldots \\ \cup &\ \langle c_m\ t_{m1} \ldots t_{mn_m} \rangle\,. \end{aligned}$$

where x is a fresh type variable, the constructors $c_1, \ldots, c_m$ are distinct, each constructor c_i is of arity n_i, and t_{ij} is a type expression possibly containing the variable x, for $1 \leq i \leq m$ and $1 \leq j \leq n_i$.

The data type definition above should be read as a shorthand for the constraint system

$$\begin{aligned} G_x = \{ &\ \langle c_i\ t_{i0}\ t_{i1} \ldots t_{in_i} \rangle \leqslant x, \\ &\ x \restriction c_i \leqslant \langle c_i\ t_{i0}\ t_{i1} \ldots t_{in_i} \rangle \mid 1 \leq i \leq m \} \\ \cup \{ &\ x \setminus \{c_1, \ldots, c_m\} \leqslant 0 \} \end{aligned}$$

where the type expressions $t_{10}, \ldots, t_{m0}$ in contra-variant position are fresh variables.

Intuitively, the first line of the definition of G_x says that the type x includes anything indicated by the data type definition, the second line that

when restricted to a particular constructor, the data type only contains values indicated by the corresponding line in the data type definition, and the third line that the values contained in the data type use no constructors except the ones in the definition.

It is straight-forward to establish that if a constraint system G_x of the form described above is added to a consistent constraint system G which does not mention x, the resulting constraint system is consistent.

Some examples. The data type definitions in the examples may be recursive, but two data type definitions may not be mutually recursive.

$$\mathsf{data}\ d_1 = \langle c_1\ t_1 \rangle$$

$$\begin{aligned} \mathsf{data}\ d_2 = {} & \langle c_1\ t_1 \rangle \\ & \cup \langle c_2\ t_2 \rangle \end{aligned}$$

$$\mathsf{data}\ d_3 = \langle c_1\ t_1\ t_3 \rangle$$

$$\mathsf{data}\ d_4 = \langle c_1\ t_4 \rangle$$

The corresponding constraint systems for d_1 and d_2:

$$\begin{aligned} G_1 = \{ & \langle c_1\ t_1 \rangle \leqslant d_1, \\ & d_1 \restriction c_1 \leqslant \langle c_i\ t_1 \rangle, \\ & d_1 \setminus c_1 \leqslant 0 \} \end{aligned}$$

$$\begin{aligned} G_2 = \{ & \langle c_1\ t_1 \rangle \leqslant d_2, \\ & \langle c_2\ t_2 \rangle \leqslant d_2, \\ & d_2 \restriction c_1 \leqslant \langle c_i\ t_1 \rangle, \\ & d_2 \restriction c_2 \leqslant \langle c_2\ t_2 \rangle, \\ & d_2 \setminus \{c_1, c_2\} \leqslant 0 \} \end{aligned}$$

Since the data type d_2 adds a case to d_1, we'd expect d_1 to be a subtype of d_2. In fact, the derivation rules for constraints do not allow us to deduce the relationship from the constraint systems, but it is easy to see that the

constraint system $G_1 \cup G_2 \cup \{d_1 \leqslant d_2\}$ is consistent while $G_1 \cup G_2 \cup \{d_2 \leqslant d_1\}$ is not. Thus a function that expects a value of type d_2 can be applied to a value of type d_1. Similarly, d_3 is a subtype of d_1 as its single constructor expression is an extension of the corresponding expression of d_1. The data type d_4 may be related to the type d_1, depending on the relationship between types t_1 and t_4.

9 Related work

Later, Kozen, Palsberg and Schwartzbach [14] showed that the problem of checking that an expression in lambda-calculus can be typed by a subtyping system could be solved in $O(n^3)$ time. Their construction involves a graph representation of the constraint system and a finite state automaton so it is rather indirect. 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. If the automaton contains a loop, the solution will correspond to a recursive type. One disadvantage with this solution is that even though the language is simple, the construction is complicated. For a more complex language, the construction would probably be impractical.

Palsberg and O'Keefe [21] described a different approach to type inference. Their constraint language more elaborate than that of Kozen et al. and contains an atomic integer type, a construction for recursive types, and the empty type. They used an algorithm based on Control Flow Analysis (CFA) [12, 25, 9] to check that a lambda term could be typed under a subtyping system and showed that the algorithm accepts all terms that type. Their construction involves a representation of recursive types as finite automata, a set of type constraints for each lambda-term, for the CFA a set of constraints for each term, and for either set of constraints a closure operator (both are somewhat similar to the construction of G^* in this paper). Thus the algorithm is complicated even though the underlying language is quite simple.

Tiuryn [29] looks at a subtyping system where the constraint language lacks empty and universal types. Unlike Kozen et al., his system also includes a set of atomic types related according to a partial order (this is the same constraint language as in this paper). In the general case the solvability problem for finite types is PSPACE-hard, but if the atomic types are organised in a disjoint set of lattices the problem can be determined in PTIME. Niehren, Priesnitz and Su [19] also consider a subtyping problem where the atomic types are related according to a partial order. By ex-

ploiting 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. Traytel, Berghofer and Nipkow [30] give a type inference algorithm for a system with structural subtyping. Their algorithm inserts coercions into a program so that it can be typed using the standard Hindley-Milner algorithm.

The papers described above all address the problem of *solving* a given constraint system, i.e., constructing a model for a constraint system (that represents a typing problem) using a given domain. The approach typically assumes a domain consisting of expressions in the constraint language and attempts to construct a model in this domain. If a model 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 for which a model cannot be constructed in this manner. Judging from previous work, it appears that trying to solve the constraint system instead of (as described in the current paper) showing its consistency adds considerable complications. Even so, not much work has been done to address the problem of typing a program by establishing consistency of a constraint system (a few exceptions are listed below).

One exception is the system described by Aiken and Wimmers [1]. Their system has union and intersection and solutions to recursive equations but no function types (thus not directly comparable to the system described in the current paper). Their algorithm computes a closure over the constraint system, and 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 which implies that it is unlikely that a practical programming language implementation could be based on this algorithm. A related system which also includes function types is presented in [2] but its complexity is not discussed.

Eifrig, Smith and Trifonov [5] describe an approach to subtyping which arguably also considers logical consistency (but this is not how the authors describe it). What is referred to as types in that paper corresponds to type expressions in the current paper, thus there is (as in the current paper) no domain of types specified. They define a closure operator on constraint systems which is very similar to the construction in this paper and show a subject-reduction property which implies, in the terminology of the current paper,

that local consistency is preserved under reduction. Thus, they guarantee a safety property for any lambda-term that has a typing. As in the current paper, their formalism can reason about programs that rely on infinite types, even though their type expressions are always finite. However, unlike the current paper which shows that any locally consistent constraint system is logically consistent (i.e., it is not possible to derive a contradiction), there is no characterisation of local consistency and thus no result corresponding to Theorem 6.13. This is of course unsatisfactory from a theoretical point of view, but it also has practical consequences. Since their proof of the subject reduction property interacts with the algorithm for checking local consistency the proof becomes rather complicated (the proof was not included in the journal version of the paper, but is available in a technical report version of the paper). If one wants to add features to the type system that affect the algorithm for checking local consistency, the proof for the subject reduction property will need to be extended which is of course a difficult undertaking, and since there is no characterisation of local consistency there is no way to know that the algorithm for checking local consistency is indeed the correct one without showing the subject reduction property. In contrast, since the current paper shows that any locally consistent constraint system is logically consistent and thus has a model it is possible to adopt Mitchell's proof almost without change.

Pottier [22] claims that Palsberg [21] gives a proof that a constraint system (which can type higher-order functions) is consistent if and only if it solvable. However, this appears to be due to a mixup as Palsberg's paper does not discuss consistency. There is another paper by Palsberg [20] (also referenced by Pottier) which shows an equivalence between consistency and solvability but in this case the programming language lacks higher-order functions and the constraint language does not include contravariant type constructors. It would certainly be very interesting to see an explicit construction of a domain of types for which all logically consistent constraint systems had a solution but to the author's best knowledge no such construction has been presented.

Many authors have defined entailment as: Given a constraint system and some constraint, is the constraint satisfied in every solution to the constraint system? (In this section, we will use the term *satisfiability entailment* to indicate this relationship.) Note that this property is distinct from entailment as it is defined in logic (which is the definition used in this paper, and which we will refer to as *logical entailment* in this section). For example, a constraint system may be consistent but unsolvable which implies that any constraint is satisfiability entailed but some constraints would not be logically entailed. Henglein and Rehof [10] consider the problem of satisfiability entailment for

a simple subtyping system. They show that the problem of determining satisfiability entailment is in general coNP-complete.

Pottier [22] gives a third definition of entailment. Given a constraint systems G and G', the first constraint system entails the second if for any constraint system H, $G' \cup H$ is consistent whenever $G \cup H$ is consistent, where consistency is defined using an algorithm, as discussed earlier in this section. Pottier gives an algorithm which attempts to check this form of entailment by constructing a proof. However, as he shows there are cases where the algorithm will fail to discover entailment. There is no discussion on the complexity of the algorithm but as the algorithm will need to traverse a large space of possible proofs the complexity appears to be high.

It has been known for a long time (the work by Pottier [23] and Flanagan and Felleisen [7] is based on this insight) that a straight-forward way to implement polymorphism is to simply copy the constraint system for every use of a let-bound variable. This is obviously inefficient and would lead to a type checker with exponential complexity. However, as shown by Mairson [15] Hindley-Milner type checking is actually DEXPTIME-hard, thus the straight-forward approach is actually asymptotically optimal. The difference between Hindley-Milner type checking and the straight-forward approach is of course that Hindley-Milner type checking relies on an efficient algorithm for computing the *principal type* of functions. Thus we obtain an optimal representation of the types of let-bound variables. In practice, this leads to an efficient analysis even though there are as Mairson showed programs which take very long to type check. In a type system with subtyping, results by Pottier [23] and Flanagan and Felleisen [7] suggest that computing an optimal representation of the constraint system associated with a let-bound variable would have a high (exponential) complexity. (Both Pottier and Flanagan relate the problem of finding an optimal representation to the problem of minimising non-deterministic finite automata.) Since the motivation for computing optimal representations of types of let-bound variables is to speed up the type checker an expensive algorithm is clearly counter-productive. Both Pottier and Flanagan conclude that a reasonable compromise is the use of faster simplification algorithms which are not guaranteed to produce optimal results and present such simplification algorithms.

Several authors have discussed the limitations of type systems in current functional programming languages with respect to algebraic data types. Wadler [32] defines the expression problem as the problem of defining a data structure so that it can simultaneously be extended with more operations and a richer data type. (He proposes a rather elaborate solution for an object-oriented programming language using parametric types and the visitor pattern.) The AST typing problem [24, 37] also concerns the ability to

extend data types, but also the ability to restrict a data type, in situations where it is known that a sub-tree is of one of a few cases. A third issue is the handling of different cases uniformly, without a dispatch. The type system presented in this paper can reason about extended or restricted data types. The third issue can probably also be managed, but this would require a detailed discussion which is outside the scope of this paper.

10 Conclusions

This paper presents a new approach to type inference for functional programming languages which considers the logical 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 algorithm for type inference is also simpler than existing solutions for subtyping systems. One important advantage of simplicity is that it makes the system easier to extend.

We show the subject reduction property which gives the usual safety guarantees associated with static typing systems, i.e., that if a program types type errors cannot occur at run-time. In our approach, the type checker checks whether the program can be typed without actually building a mapping from program variables to types. Thus type information is not immediately available (at least not for complex types). To remedy this, we extended the algorithm for type checking to also determine entailment, that is, whether two arbitrary type expressions are related by the subtype order. It turns out that this information can be computed without an increase in complexity.

A common problem with subtyping systems for languages based on lambda calculus is that it is hard to demonstrate the advantages of subtyping if one stays close to lambda calculus. A practical application of subtyping might be the definition of two algebraic data types where one is a subtype of the other. But this implies that some constructors might belong to both the more general and the more specific data type while others would only be used in the more general type. Also, one would like support for determining at run-time if a value that is known to be of the more general data type is in fact of the more specific type (this would hold if it uses no constructor of the more general type). To solve this we introduce constructors in our language. The basic idea is not new; many programming languages use similar solutions, but a new idea is that the constructors build both values and types. Thus, any collection of constructors constitutes a type. We also extend the language with a relatively new concept we call open case expression (this form

of expressions have only received minimal attention in the literature) and we show how open case expressions allows a program to distinguish between values of a more general and a more specific type.

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] H. P. Barendregt. *The Lambda Calculus, Its Syntax and Semantics*. Number 103 in Studies in Logic and the Foundations of Mathematics. North Holland, 1984.

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

[5] Jonathan Eifrig, Scott Smith, and Valery Trifonov. Type inference for recursively constrained types and its application to OOP. *Electronic Notes in Theoretical Computer Science*, 1:132–153, 1995.

[6] Cormac Flanagan. Hybrid type checking. In *POPL'06*, pages 245–256, New York, NY, USA, 2006. ACM.

[7] Cormac Flanagan and Matthias Felleisen. Componential set-based analysis. *ACM Trans. Program. Lang. Syst.*, 21(2):370–416, March 1999.

[8] Kurt Gödel. Die Vollständigkeit der Axiome des logischen Funktionenkalküls. *Monatshefte für Mathematik*, 37(1):349–360, 1930. Translation to English: The completeness of the axioms of the functional calculus of logic, *Kurt Gödel: Collected Works: Publications 1929-1936.* Oxford University Press, 1986, pp. 102-123.

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

[10] 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*, pages 352–361. IEEE Computer Society Press, 1997.

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

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

[13] Kenneth Knowles and Cormac Flanagan. Hybrid type checking. *ACM Trans. Program. Lang. Syst.*, 32(2):6:1–6:34, February 2010.

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

[15] Harry G. Mairson. Deciding ML typability is complete for deterministic exponential time. In *POPL'90*, pages 382–401, New York, NY, USA, 1990. ACM.

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

[17] John C. Mitchell. Coercion and type inference. In *Principles of Programming Languages*, pages 175–185. ACM, 1984.

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

[19] Joachim Niehren, Tim Priesnitz, and Zhendong Su. Complexity of subtype satisfiability over posets. In *Programming Languages and Systems*, pages 357–373, 2005.

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

[21] Jens Palsberg and Patrick O'Keefe. A type system equivalent to flow analysis. *ACM Toplas*, 17(4):576–599, July 1995.

[22] François Pottier. Simplifying subtyping constraints. In *ICFP'96*, pages 122–133, New York, NY, USA, 1996. ACM.

[23] François Pottier. Simplifying subtyping constraints: a theory. *Information and Computation*, 170(2):153–183, 2001.

[24] Jonathan S. Shapiro. The AST typing problem. Lambda the ultimate weblog, `http://lambda-the-ultimate.org/node/4170`, December 2010.

[25] Olin Shivers. Control flow analysis in Scheme. In *PLDI'88*, pages 164–174, 1988.

[26] Jeremy G Siek and Ronald Garcia. Interpretations of the gradually-typed lambda calculus. In *Proceedings of the 2012 Annual Workshop on Scheme and Functional Programming*, pages 68–80. ACM, 2012.

[27] Jeremy G Siek and Walid Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, volume 6, pages 81–92, 2006.

[28] Satish Thatte. Type inference with partial types. In *Automata, Languages and Programming*, pages 615–629. Springer, 1988.

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

[30] Dmitriy Traytel, Stefan Berghofer, and Tobias Nipkow. Extending Hindley-Milner type inference with coercive structural subtyping. In *Programming Languages and Systems*, pages 89–104. Springer, 2011.

[31] Dirk van Dalen. *Logic and structure, Fourth edition.* Springer-Verlag, 2008.

[32] Philip Wadler. The expression problem. Posted on the Java genercity mailing list, November 1998.

[33] Philip Wadler and Robert Bruce Findler. Well-typed programs can't be blamed. In *Programming Languages and Systems*, pages 1–16. Springer, 2009.

[34] Mitchell Wand. A simple algorithm and proof for type inference. *Fundamenta Informaticae*, 10(2):115–121, 1987.

[35] Mitchell Wand and Patrick O'Keefe. On the complexity of type inference with coercion. In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, FPCA '89, pages 293–298, New York, NY, USA, 1989. ACM.

[36] Tobias Wrigstad, Francesco Zappa Nardelli, Sylvain Lebresne, Johan Östlund, and Jan Vitek. Integrating typed and untyped code in a scripting language. In *POPL'10*, pages 377–388, New York, NY, USA, 2010.

[37] Edward Z. Yang. The AST typing problem. Weblog entry, `http://blog.ezyang.com/2013/05/the-ast-typing-problem`, May 2013.