Constraint-set Satisfiability for Overloading

Carlos Camarão
Univ. Fed. de Minas Gerais, DCC-ICEX
Belo Horizonte 31270-010, Brasil
camarao@dcc.ufmg.br

Lucília Figueiredo
Univ. Federal de Ouro Preto, DECOM-ICEB
Ouro Preto 35400-000, Brasil
lucilia@dcc.ufmg.br

Cristiano Vasconcellos
Pontifícia Univ. Católica do Paraná, DI-CCET
Curitiba 80215-901, Brasil
damiani@ppgia.pucpr.br

ABSTRACT

This article discusses the problem of *constraint-set satisfiability* (*CS-SAT*) — that is, the problem of determining whether a given constraint-set is satisfiable in a given typing context — in the context of systems with support for overloading and parametric polymorphism. The paper reviews previous works on constraint-set satisfiability, showing that overloading policies used in order to guarantee decidability of *CS-SAT* have been generally too restrictive. An algorithm is proposed that does not impose a severe restriction on possible overloadings and decides *CS-SAT* in an expectedly vast majority of cases of practical interest. In cases for which satisfiability cannot be decided, a configurable limit on the number of iterations is used in order to guarantee termination.

1. INTRODUCTION

A language or type system that supports overloading and parametric polymorphism — a combination which we prefer to call *constrained polymorphism*, instead of the more usual term *ad-hoc polymorphism* — makes use of an overloading policy to restrict overloading, in order to obtain a balance between performing type inference (or type checking) in a reasonably efficient way, on one hand, and considering as valid a large set of programs that make use of overloading, on the other hand. Some early type systems adopted a rather simple context-independent overloading policy [17, 22], which restricts overloading so that each overloaded function f must be such that, for each application of f to an expression e, the decision of which function to be applied can be determined according to the type of e. With context-dependent overloading, on the other hand, this decision can be made by considering the (program) context where the application of f to e occurs.

A context-independent overloading policy is adopted nowadays in several mainstream programming languages, like e.g. C++ and Java, for methods defined in the same class (that is, disregarding the fact that dynamic binding of names to methods can be seen as a form of overloading resolution). While such an approach enables simple solutions to problems related to overloading (overloading resolution in particular), it is rather restrictive. For example, constant symbols cannot be overloaded, neither can a function name such as *read*, with definitions having types that are instances of the polymorphic type $\forall a.\, String \rightarrow a$. A context-dependent overloading policy, on the other hand, allows such definitions. For example, the type of *read* in $\lambda x.\, read\, x$ `==` `"a string"` can be determined to be $String \rightarrow String$.

Many type systems for overloading have adopted a less restrictive, context-dependent overloading policy. These include system `CT` [1], Haskell's type classes [21, 13, 10] and other related systems [33, 2, 27, 14, 7, 5, 9, 8, 25, 28]. The constraint-set satisfiability problem (*CS-SAT*) is to determine, given a constraint-set κ and a typing context Γ, whether κ is satisfiable in Γ. It is an important problem in these systems, for which there is no known widely accepted solution. In this paper we present an algorithm for the solution of *CS-SAT*, without imposing a restrictive overloading policy.

The paper is organized as follows. Section 2 gives an informal overview of system `CT`, a type system designed for the support of constrained polymorphism. Section 3 introduces basic notations and terminology, including definitions of what constitutes an overloading policy and when a set of assumptions is considered as a typing context. *CS-SAT* is defined in Section 4. Section 5 reviews overloading policies. Our solution is presented in Section 6. Section 7 presents a significant optimization for checking constraint-set satisfiability. Section 8 concludes.

Due to space reasons, we cannot discuss in this paper several other important topics related to overloading, such as constraint-set simplification and ambiguity, and in particular we only include an informal description of type system `CT`.

2. OVERVIEW OF SYSTEM CT

Type system `CT` is an extension of the Damas-Milner type system [3] for the support of overloading. We assume a set of types of terms of a language that is basically core-ML [18, 3, 19, 20] extended with the possibility of introducing overloaded definitions. Thus, typing contexts may have more

PPDP Verona, Italy, August 24 - 26, 2004

than one assumption for the same variable, and the set of assumptions for a variable may be extended in let-bindings.

Types of expressions are constrained polymorphic types. A set of constraints κ is a (possibly empty) set of pairs $o : \tau$, where o is an overloaded name and τ is a simple type; a constrained polymorphic type is written as $\forall\alpha_1. \ldots. \forall\alpha_n. \kappa. \tau$, where $n \geq 0$ and α_i is a type variable, for $i = 1, \ldots, n$.

A typing context Γ is formed by a finite number of type assumptions $x : \sigma$, where x is a name (or symbol) and σ is a constrained polymorphic type. An overloaded symbol has more than one type assumption in Γ. The set of valid type assumptions for an overloaded symbol in a typing context is defined by an overloading policy, as discussed in Sections 3 and 5.

The *principal type* of an expression e, in a given typing context Γ, is a minimal (or *the* least, if we consider types to be equivalent up to a proper renaming of type variables) type that is general enough to represent the set of all types that can be derived for e in Γ. The types represented by the principal type of an expression are called its *instances*. The principal type of an overloaded symbol o is obtained by quantifying over the type variables of the least common generalization (*lcg*) of the set of types in assumptions for o in Γ. The use of *lcg* is fundamental for the computation of unique principal types, while allowing typing contexts to be stepwisely extended with overloaded definitions. The use of *lcg* in the type system should not be surprising, given that *principal* means *minimal* and general enough to represent the set of all derivable types. Related type systems that do not (need to) use *lcg* either have principal types of overloaded symbols fixed a priori, in a global typing context (cf. e.g. [21, 13, 22, 9]), or introduce "multiple principal types" [26, 27].

For any simple types τ, τ' and any substitution S, let $\tau \leq_S \tau'$ hold if $S(\tau) = \tau'$, and let $\tau \leq \tau'$ hold if there exists S such that $\tau \leq_S \tau'$. Let also $\tau \leq_S \mathbb{T}$ hold, for some set of simple types $\mathbb{T}$, if $\tau \leq_S \tau'$, for all $\tau' \in \mathbb{T}$. Analogously, let $\tau \leq \mathbb{T}$ hold if there exists S such that $\tau \leq_S \mathbb{T}$ holds. $lcg(\tau, \mathbb{T})$ is defined to hold if $\tau \leq \mathbb{T}$ holds and, whenever $\tau' \leq \mathbb{T}$, we have that $\tau' \leq \tau$.[1] A function that computes a least common generalization of a set of simple (non-quantified) types is given in Section 3.

Example 1. Consider that the assumptions for `(==)` in a typing context called $\Gamma_{(==)}$ are:

$$\texttt{(==)} : \mathit{Int} \to \mathit{Int} \to \mathit{Bool}$$
$$\texttt{(==)} : \mathit{Float} \to \mathit{Float} \to \mathit{Bool}$$

The following types can be derived for `(==)` in this typing context:

$$\texttt{(==)} : \mathit{Int} \to \mathit{Int} \to \mathit{Bool}$$
$$\texttt{(==)} : \mathit{Float} \to \mathit{Float} \to \mathit{Bool}$$
$$\texttt{(==)} : \forall a.\, \{\texttt{(==)} : a \to a \to \mathit{Bool}\}.\, a \to a \to \mathit{Bool}$$

The last one is the principal type of `(==)` in $\Gamma_{(==)}$. It can be instantiated to types of the form $\{\texttt{(==)} : \tau \to \tau \to \mathit{Bool}\}.\, \tau \to \tau \to \mathit{Bool}$, for which the constraint is *satisfiable* in Γ — in this particular case, τ can be either *Int* or *Float* or α, for some type variable α. If τ is *Int* or *Float*, the set of constraints can be simplified to an empty constraint-set (i.e. the constraints can be removed).

Example 2. Consider the following definition of function *ins*, that uses `(==)`:

```
ins a []     = [a]
ins a (b:x) = if a==b then b:x else b:ins a x
```

The principal type of a recursive let-binding corresponding to this definition, obtained by using the least common generalization of types in the assumptions for `(==)` in $\Gamma_{(==)}$, is the following:

$$\forall a.\, \{\texttt{(==)} : a \to a \to \mathit{Bool}\}.\, a \to [a] \to [a]$$

In a typing context with a type assumption for *ins* corresponding to this definition, in addition to the type assumptions in $\Gamma_{(==)}$, this type tells us that *ins* can be used in any context where an expression of type $\{\texttt{(==)} : \tau \to \tau \to \mathit{Bool}\}.\, \tau \to [\tau] \to [\tau]$ can be used, if constraint-set $\{\texttt{(==)} : \tau \to \tau \to \mathit{Bool}\}$ is satisfiable in this typing context. Additional definitions of `(==)` can be visible in this context, as for example a definition of `(==)` with type $\mathit{Char} \to \mathit{Char} \to \mathit{Bool}$ and, in this case, *ins* could also be used with type $\mathit{Char} \to [\mathit{Char}] \to [\mathit{Char}]$.

Functions may also be overloaded to operate over distinct type constructors, as in the following example.

Example 3. We assume in this example distinct definitions of function *ins*, corresponding to operations for inserting elements in lists and trees, originating the following type assumptions:

$$\mathit{ins} : \forall a.\, \{\texttt{(==)} : a \to a \to \mathit{Bool}\}.\, a \to [a] \to [a],$$
$$\mathit{ins} : \forall a.\, \{\texttt{(==)} : a \to a \to \mathit{Bool}\}.\, a \to \mathit{Tree}\ a \to \mathit{Tree}\ a$$

In a typing context containing these assumptions, say $\Gamma_{\texttt{ins}}$, the following type can be derived for *ins* (where c is a constructor variable):

$$\forall a.\, \forall c.\, \{\mathit{ins} : a \to c\ a \to c\ a\}.\, a \to c\ a \to c\ a$$

[1] $\leq$ can be extended to a partial order on all polymorphic constrained types, for which quantification is antimonotonic, that is, if $\tau \leq \tau'$ then $\sigma' \leq \sigma$ (read: σ is more general than σ'), where σ and σ' are obtained by quantifying all type variables in τ and τ', respectively.

Note that this type does not explicitly contain a constraint on `(==)`. This constraint is automatically recovered from the constraints on types of the assumptions for *ins* in $\Gamma_{\texttt{ins}}$, therefore implicitly creating a hierarchy of dependencies between overloaded symbols.

A new definition of an overloaded symbol must not necessarily have a type that is an instance of the least common generalization of types given by previous definitions. Instead, any new definition may imply the assignment of a more general type than that computed according to previous definitions. This is illustrated by the following example.

Example 4. Consider that we also want to overload *ins* with definitions that take a comparison operator as an argument, operating on ordered lists and trees and originating the following type assumptions, additionally to those in $\Gamma_{\texttt{ins}}$ of Example 3:

$$ins : \forall a.\, (a \to a \to Bool) \to a \to [a] \to [a]$$
$$ins : \forall a.\, (a \to a \to Bool) \to a \to Tree\ a \to Tree\ a$$

The type derived for *ins*, in a typing context also including these assumptions, is shown below:

$$\forall a.\, \forall b.\, \forall c.\, \{ins : a \to b \to c\}.\ a \to b \to c$$

a		b		c
a	$\to$	$[a]$	$\to$	$[a]$
a	$\to$	$Tree\ a$	$\to$	$Tree\ a$
$a \to a \to Bool$	$\to$	a	$\to$	$[a] \to [a]$
$a \to a \to Bool$	$\to$	a	$\to$	$Tree\ a \to Tree\ a$

$$lcg(a, \{a, a, a \to a \to Bool, a \to a \to Bool\})$$
$$lcg(b, \{[a], Tree\ a, a, a\})$$
$$lcg(c, \{[a], Tree\ a, [a] \to [a], Tree\ a \to Tree\ a\})$$

3. NOTATION

Types have the following context-free syntax:

Definition 1.

Simple Types	τ	$::= C\ \tau_1 \ldots \tau_n \mid \alpha\ \tau_1 \ldots \tau_n$	$(n \geq 0)$
Constraints	κ	$::= \{o : \tau\} \mid \kappa \cup \kappa'$	
Types	σ	$::= \tau \mid \kappa.\, \tau \mid \forall \alpha.\sigma$	

Meta-variables α, β, a, b and c are used for type and constructor variables. We assume that there is a given set of type constructors.

We use $\forall \bar{\alpha}.\, \kappa.\, \tau$ as an abbreviation for $\forall \alpha_1. \cdots \forall \alpha_n.\, \kappa.\, \tau$, for some $n \geq 0$. Similarly, $\overline{\kappa.\, \tau}$ denotes $\{\kappa_i.\, \tau_i\}^{i=1..n}$, for some $n \geq 0$, and analogously for $\bar{\tau}$, $\bar{\sigma}$ etc. We assume that $\forall \bar{\alpha}.\emptyset.\tau = \forall \bar{\alpha}.\tau$.

We assume, for simplicity, that term variables ($x \in X$) are divided into let-bound ($o \in O$) and lambda-bound ($u \in U$). Meta-variable A is used to denote a set of type assumptions, for which it is assumed that if $x \in A$ and $x \in U$ then $\sigma = \tau$, for some simple type τ. For any set of type assumptions $A = \{x_i : \sigma_i\}^{i=1..n}$, we define $dom(A) = \{x_i\}^{i=1..n}$. Letting $\{x : \sigma_i\}^{i=1..n}$ be the (possibly empty) set of all assumptions for x in A, we define $A(x) = \{\sigma_i\}^{i=1..n}$.

The set of free type variables of type σ is defined as usual and denoted by $tv(\sigma)$. $tv(\kappa)$ and $tv(A)$ are defined similarly, taking into account the types occurring in κ and A, respectively. We use $tv(t_1, \ldots, t_n)$ as an abbreviation for $tv(t_1) \cup \ldots \cup tv(t_n)$. Type σ is called *closed* if $tv(\sigma) = \emptyset$.

An overloading policy, denoted by meta-variable ρ, is a predicate on a set of type assumptions that specifies whether $A(x)$ is a valid set of types of definitions of x, for each symbol x. In this paper we use the following.

Definition 2 (Typing context). *A set of type assumptions A is a* typing context *if $\rho(A)$ holds, for some overloading policy ρ.*

unify(E) is assumed to give the most general unifying substitution for the set of pairs of type expressions E, usually written as a set of type equations [23]. A definition of *unify* can be given as usual (see e.g. [20, page 774]).

A substitution S is a function from type or constructor variables to simple types or type constructors, respectively. The identity substitution is denoted by *id*. As usual, $\circ$ denotes function composition. $S\sigma$ represents the capture-free[2] operation of substituting $S(\alpha)$ for each free occurrence of type variable α in σ. This operation is extended to constraints and typing contexts in the usual manner, and juxtaposition is right-associative, so that, for example, $S\, S'\, \sigma$ denotes $S(S'(\sigma))$. We define $dom(S) = \{\alpha \mid S(\alpha) \neq \alpha\}$. For ease of notation, it will be often convenient to use a finite mapping notation for substitutions. In this case, we write $S = \{(\alpha_j \mapsto \tau_j)\}^{j=1..m}$ to denote the substitution such that $dom(S) = \{\alpha_j\}^{j=1..m}$ and $S(\alpha_j) = \tau_j$, for $j = 1, \ldots, m$.

For any function f, $f \dagger \{a_i \mapsto \tau_i\}^{i=1..n}$ denotes the function f' such that $f'(x) = f(x)$, if $x \notin \{a_i\}^{i=1..n}$, and $f'(a_i) = \tau_i$, for $i = 1, \ldots, n$. We define: $\sigma[\tau/\alpha] = (id \dagger \{\alpha \mapsto \tau\})\sigma$.

We define that $inst(\sigma, \kappa.\, \tau)$ holds if $\sigma = \forall \bar{\alpha}.\, \kappa'.\, \tau'$ and $\kappa.\, \tau = (\kappa'.\, \tau')[\bar{\tau}/\bar{\alpha}]$, for some $\bar{\tau}$. Analogously, $gen(\kappa.\, \tau, \sigma)$ is defined to hold if $\sigma = \forall \bar{\beta}.\, \kappa.\, \tau[\bar{\beta}/\bar{\alpha}]$, for some $\bar{\beta}$, and $\bar{\alpha} = tv(\kappa.\, \tau)$. We also use $\overline{\overline{\kappa.\, \tau}}$ to denote any generalisation of $\kappa.\, \tau$, i.e. any member of $\{\sigma \mid gen(\kappa.\, \tau, \sigma)\}$. Similarly for $\overline{\overline{\kappa}}$, that is, letting $\kappa = \{o_i : \tau_i\}^{i=1..n}$, we have that $\overline{\overline{\kappa}} = \{o_i : \overline{\overline{\tau_i}}\}^{i=1..n}$, and $\overline{\overline{\kappa}}$ can also be written as $\{\|o_1 : \tau_1, \ldots, o_n : \tau_n\|\}$.

A function that computes a least common generalization of a set of simple types is given in Figure 1.

For ease of notation, we make a simplification and consider *lcg* as a function over a set of types, by choosing an arbitrary

[2]The operation of applying substitution S to σ is capture-free if $tv(S\sigma) = tv(S(tv(\sigma)))$, where application of substitution S to a set of type variables $\{\alpha_i\}^{i=1..n}$ is given by $\{S(\alpha_i)\}^{i=1..n}$.

$$lcg(\mathbb{T}) = \tau \quad \text{where } (\tau, \mathcal{S}) = lcg'(\mathbb{T}, \emptyset), \text{ for some } S$$

$$lcg'(\{\tau\}, S) = (\tau, S)$$

$$lcg'(\{\nu\,\tau_1 \ldots \tau_n,\ \nu'\,\tau'_1 \ldots \tau'_m\}, S) =$$
$$\textbf{if } S(\alpha) = (\nu\,\tau_1 \ldots \tau_n,\ \nu'\,\tau'_1 \ldots \tau'_m) \text{ for some } \alpha \textbf{ then } (\alpha, S) \textbf{ else}$$
$$\textbf{if } n \neq m \textbf{ then } (\alpha', S \dagger \{\alpha' \mapsto (\nu\,\tau_1 \ldots \tau_n,\ \nu'\,\tau'_1 \ldots \tau'_m)\}) \text{ where } \alpha' \text{ is a fresh type variable}$$
$$\textbf{else } \nu_0\,\tau''_1\ \ldots\ \tau''_n \text{ where } (\nu_0, S_0) = \begin{cases} (\nu, S) & \text{if } \nu = \nu' \\ (\alpha, S \dagger \{\alpha \mapsto (\nu, \nu')\}) & \text{otherwise, where } \alpha \text{ is fresh} \end{cases}$$
$$(\tau''_i, S_i) = lcg'(\{\tau_i, \tau'_i\}^{i=1..n}, S_{i-1}),\ \text{for } i = 1, \ldots, n$$

$$lcg'(\{\tau_1, \tau_2\} \cup \mathbb{T}, S) = lcg'(\{\tau, \tau'\}, S') \quad \text{where } \begin{array}{ll} (\tau, S_0) & = lcg'(\{\tau_1, \tau_2\}, S) \\ (\tau', S') & = lcg'(\mathbb{T}, S_0) \end{array}$$

Figure 1: Least common generalization of a set of types

representative of the equivalence class of types that are least common generalizations of $\{\tau_i\}^{i=1..n}$, where τ is equivalent to τ' if they are equal except for renaming of fresh type variables. Meta-variable ν is used to denote a type or a constructor variable.

An extra parameter is used in the definition of lcg' so that, for example, the least common generalization of the set of types $\{\alpha_1 \to \beta_1 \to \alpha_1,\ \alpha_2 \to \beta_2 \to \alpha_2\}$ is $\alpha \to \beta \to \alpha$, for some fresh type variables α, β (and not, say, $\alpha \to \beta \to \alpha'$).

We also need a definition of lcg between a set of types and a set of sets of types, defined as follows. Let $\{\tau_i\}^{i=1..n} \leq_S \{\{\tau_{ij}\}^{i=1..n}\}^{j=1..m}$ hold if $S\tau_i = \tau_{ij}$, for $i = 1, \ldots, n, j = 1, \ldots, m$, and $\{\tau_i\}^{i=1..n} \leq \{\{\tau_{ij}\}^{i=1..n}\}^{j=1..m}$ hold whenever $\{\tau_i\}^{i=1..n} \leq_S \{\{\tau_{ij}\}^{i=1..n}\}^{j=1..m}$ holds, for some S.[3]

$lcg(\{\tau_i\}^{i=1..n}, \{\{\tau_{ij}\}^{i=1..n}\}^{j=1..m})$ is defined to hold whenever the following conditions hold:

1. $\{\tau_i\}^{i=1..n} \leq \{\{\tau_{ij}\}^{i=1..n}\}^{j=1..m}$;
2. for any set of types $\{\tau'_i\}^{i=1..n}$ such that $\{\tau'_i\}^{i=1..n} \leq \{\{\tau_{ij}\}^{i=1..n}\}^{j=1..m}$, we have that $\tau'_i \leq \tau_i$, for $i = 1, \ldots, n$.

Letting $\mathbb{S}$ be a set of substitutions, $\cap\mathbb{S}$ denotes any substitution R such that, for all α, $lcg(R(\alpha), \{S(\alpha) \mid S \in \mathbb{S}\})$ holds.

4. CONSTRAINT-SET SATISFIABILITY

Informally speaking, the occurrence of a constraint-set κ in the principal type of an expression e indicates that it is possible for this expression to occur in some context, as a subexpression of another expression, whose principal type has none of the constraints in κ, because the use of e in this context has enabled overloading of all symbols in κ to be resolved. This elimination of constraints can occur when the types of overloaded symbols in κ have been instantiated in such a way that it is possible to determine which of the definitions of each overloaded symbol should be used for the evaluation of e.

It should be noted that it is possible for constraints $o_i : \tau_i$, $i = 1, \ldots, n$, in a constraint-set κ, to be separately satisfiable in a given typing context Γ, whereas κ is not. Consider:

Example 5. Let $\Gamma_1 = \{\ f : Int \to Int,\quad f : Float \to Float,\quad o : Char,\quad o : Bool\}$

Constraints $f : \alpha \to \alpha$ and $o : \alpha$ are both satisfiable in Γ_1, when considered separately, whereas $\kappa_1 = \{f : \alpha \to \alpha, o : \alpha\}$ is not.

Geoffrey Smith and Dennis Volpano have examined the *CS-SAT* problem [26, 32, 27, 30, 31], stated as whether, given a constraint set κ and a typing context Γ, there exists a substitution S such that $\Gamma \vdash S\kappa$ is provable (Smith and Volpano have examined the *CS-SAT* problem for global overloading, as we also do in this paper). For any κ, the provability of $\Gamma \vdash \kappa$ is defined as equivalent to the provability of $\Gamma \vdash x : \tau$, for all $x : \tau \in \kappa$. For arbitraries Γ and κ, *CS-SAT* has been shown to be undecidable [26, 32].

As pointed out by Mark Jones [13], a definition of *CS-SAT* should be independent on provability in a type system. In his work, typability is separated from so-called predicate entailment (which in our case means constraint-set satisfiability). Mark Jones did not, however, fully considered the question of constraint-set satisfiability, only examining the problem in general terms, in [15].

We give below a revised definition of *CS-SAT* that is independent on provability in a type system, being given only in terms of the given inputs, namely a set of constraints and a set of type assumptions. This may help in reasoning about the problem and in establishing connections with other problems. Let (where 2^Γ denotes the powerset of Γ):

[3]Note that, for $\{\tau_i\}^{i=1..n} \leq \{\{\tau_{ij}\}^{i=1..n}\}^{j=1..m}$ to hold, it is not sufficient that $\tau_i \leq \tau_{ij}$ holds, for all $i = 1, \ldots, n, j = 1, \ldots, m$. For example, $\{\alpha \to \alpha, \alpha\} \not\leq \big\{\{Int \to Int, Bool\}, \{Int \to Int, Char\}\big\}$, although $\alpha \to \alpha \leq Int \to Int$, $\alpha \leq Bool$, $\alpha \to \alpha \leq Int \to Int$ and $\alpha \leq Char$ hold.

DEFINITION 3 (CONSTRANT-SET SATISFIABILITY).

$$\frac{\begin{array}{c}\{\{o_i : \sigma_{ij}\}^{i=1..n}\}^{j=1..m} \subseteq 2^{\Gamma} \\ \textit{for } i = 1,\ldots,n,\ j = 1,\ldots,m : \textit{inst}(\kappa_{ij}.\tau_{ij}, \sigma_{ij}) \\ \textit{for } j = 1,\ldots,m : \Gamma \models \bigcup_{i=1..n} \kappa_{ij} \\ \textit{lcg}(\{\tau_i\}^{i=1..n}, \{\{\tau_{ij}\}^{i=1..n}\}^{j=1..m})\end{array}}{\Gamma \models \{o_i : \tau_i\}^{i=1..n}} \quad (\texttt{sat})$$

The proof system "works nicely for recursive constraints". For example, to prove satisfiability of constraint-set $\{(\texttt{==}) : [[\alpha]] \to [\alpha] \to \textit{Bool}\}$ in $\Gamma_{(\texttt{==})}$ of Example 1 — that is, to show that $\Gamma_{(\texttt{==})} \models \{(\texttt{==}) : [\alpha] \to [\alpha] \to \textit{Bool}\}$ is provable, we take $\textit{lcg}(\{[\alpha] \to [\alpha] \to \textit{Bool}\}, \{\{[\alpha] \to [\alpha] \to \textit{Bool}\}\})$ and prove satisfiability of $\{(\texttt{==}) : \alpha \to \alpha \to \textit{Bool}\}$ in $\Gamma_{(\texttt{==})} - \{\forall\alpha.\{(\texttt{==}).\alpha \to \alpha \to \textit{Bool}.[\alpha] \to [\alpha] \to \textit{Bool}\}$, by taking $\textit{lcg}(\{\alpha \to \alpha \to \textit{Bool}\}, \{\{\textit{Int} \to \textit{Int} \to \textit{Bool}, \textit{Float} \to \textit{Float} \to \textit{Bool}\}\})$.[4]

If $\{o_i : \tau_i\}^{i=1..n} = \emptyset$, rule (sat) is (vacuously) equivalent to the axiom $\Gamma \models \emptyset$.

We can now give:

DEFINITION 4 (*CS-SAT*). *The CS-SAT problem is to determine, given a typing context* Γ *and a constraint-set* κ*, whether or not* $\Gamma \models \kappa$ *is provable.*

Example 6. Let $\Gamma_{\texttt{f}} = \{\ f : \textit{Int} \to \textit{Int},\ f : \textit{Int} \to \textit{Float},\ f : \textit{Float} \to \textit{Float}\ \}$

According to rule (sat), constraints $\{f : \textit{Int} \to \textit{Int}\}$, $\{f : \textit{Int} \to \beta\}$, $\{f : \textit{Int} \to \textit{Float}\}$, $\{f : \alpha \to \beta\}$ and $\{f : \textit{Float} \to \textit{Float}\}$ are satisfiable in $\Gamma_{\texttt{f}}$, but not, for example, constraint $\{f : \textit{Float} \to \beta\}$, because this type is not a least common generalization of simple types in assumptions for f in $\Gamma_{\texttt{f}}$ (reflecting the fact that if f is used in a program context represented by $\Gamma_{\texttt{f}}$ in an expression where its argument has type *Float*, then its result must be of type *Float*).

[4]A closed world approach to overloading, as supported by system CT, is such that:

1. Any definition of a variable x cannot be placed in a context where there exists no definition of an overloaded variable used in the definition of x. System CT could be extended to support also an open world approach for overloading, but such an extension is outside the scope of this paper. Without such an extension, a definition of, say, (==) for lists is not typable if placed in a context where there exists no definition for (==) for the list elements.
2. If a definition of a variable x that uses another variable y is placed in a context where there exists a single definition of y, then the type of x has no constraint on y. Here, again, this is not the case if support for treating y as in an open world is provided. Note, however, the special case of a recursive definition, where $y = x$. For example, a recursive definition of (==) for lists that is placed in a context in which there exists a single distinct definition of (==) for, say, integers, would have a polymorphic type (namely $\forall\alpha.\{(\texttt{==}) : \alpha \to \alpha \to \textit{Bool}\}.\ [\alpha] \to [\alpha] \to \textit{Bool}$).

Example 7. Consider $\Gamma = \Gamma_{\texttt{f}} \cup \Gamma_{o_1} \cup \Gamma_{o_2} \cup \Gamma_{o_3}$, where $\Gamma_{\texttt{f}}$ is as in Example 6 and

$$\begin{array}{l}\Gamma_{o_1} = \{o_1 : \textit{Bool},\ o_1 : \textit{Char}\} \\ \Gamma_{o_2} = \{o_2 : \textit{Float},\ o_2 : \textit{Char}\} \\ \Gamma_{o_3} = \{o_3 : \textit{Int},\ o_3 : \textit{Float}\}\end{array}$$

Constraint-sets $\{o_1 : \alpha\}$, $\{o_2 : \alpha\}$, $\{o_3 : \alpha\}$ and $\{f : \alpha \to \beta\}$ are satisfiable in Γ. We have also that:

1. a constraint-set $\{f : \tau,\ o_1 : \tau'\}$ is satisfiable in Γ only if $tv(\tau) \cap tv(\tau') = \emptyset$;
2. $\{f : \textit{Float} \to \textit{Float},\ o_2 : \textit{Float}\}$ is the only constraint-set of the form $\{f : \tau' \to \tau,\ o_2 : \tau'\}$ satisfiable in Γ. In system CT, overloading is said to be *resolved* for any satisfiable constraint with an empty set of type variables. This constraint can be "removed" from the constraint-set in which it appears.
3. all constraint-sets of the form $\{f : \tau' \to \tau,\ o_3 : \tau'\}$ satisfiable in Γ are (up to renaming of type variables α and β): $\{f : \textit{Int} \to \textit{Int},\ o_3 : \textit{Int}\}$, $\{f : \textit{Float} \to \textit{Float},\ o_3 : \textit{Int}\}$, $\{f : \textit{Float} \to \textit{Float},\ o_3 : \textit{Float}\}$, $\{f : \textit{Int} \to \beta,\ o_3 : \textit{Int}\}$ and $\{f : \alpha \to \beta,\ o_3 : \alpha\}$.

Example 8. Consider now $\Gamma = \Gamma_{\texttt{f}} \cup \Gamma_{o_1} \cup \Gamma_{\texttt{g}}$, where:

$$\begin{array}{l}\Gamma_{\texttt{g}} = \{\ g : \forall\alpha.\{f : \alpha \to \alpha\}.\ [\alpha] \to [\alpha], \\ \qquad\quad g : \forall\alpha.\{f : \alpha \to \alpha\}.\ \textit{Tree}\ \alpha \to \textit{Tree}\ \alpha, \\ \qquad\quad o_2 : \forall\beta.\{o_1 : \beta\}.\ [\beta]\ \}\end{array}$$

$\Gamma_{\texttt{g}} \models \{g : [\alpha] \to [\alpha],\ o_2 : [\alpha]\}$ cannot be proved, since this involves proving $\Gamma_{\texttt{g}} \models \{f : \alpha_1 \to \alpha_1,\ o_1 : \alpha_1\}$ (for some fresh type variable α_1), which cannot be proved because there is no $\mathbb{T} \in 2^{\Gamma(f) \cup \Gamma(o_1)}$ such that $\textit{lcg}(\{a_1 \to a_1,\ a_1\}, \mathbb{T})$ holds.

5. *CS-SAT* AND OVERLOADING POLICIES

Without any restrictions on the assumption set, *CS-SAT* has been shown to be undecidable[26, 32]. Type systems for overloading have explored since then a number of overloading policies with restrictions on types of overloaded symbols. We discuss some of these policies below.

Let global overloading in a set of type assumptions A be characterized by:[5]

DEFINITION 5.

$$\textit{global}(A) = ((o : \sigma) \in A \textit{ and } \#A(o) > 1 \textit{ imply } tv(\sigma) = \emptyset)$$

[5]In fact, the overloading policy given by *global* is slightly more general than what we should expect from the name "global overloading", since *global* allows overloaded definitions to occur in inner scopes, as long as their types do not contain free (lambda-bound) type variables.

Global context-independent overloading (as defined for example in System O[22], and also in languages like C++ for methods defined in a single class) can be characterized by the context-independent overloading policy defined by:

$\rho_{\mathtt{ci}}(A)$ holds if *global*(A) holds and $\Big(\big(\{o{:}\sigma,\ o{:}\sigma'\} \subseteq A,$
$\sigma = \forall\bar{\alpha}.\,\kappa.\,C\ \bar{\tau} \to \tau,\ \sigma' = \forall\bar{\alpha}'.\,\kappa'.\,C'\ \bar{\tau}' \to \tau'$
(for some $\bar{\tau}, \bar{\tau}', \kappa, \kappa', C, C'$) and $\sigma' \neq \sigma\big)$ imply $C \neq C'\Big)$

Early work on context-dependent overloading (e.g. [17, 33]) did not consider the problem of constraint-set satisfiability. As a consequence, expressions with non-satisfiable constraint-sets could be well-typed. For example, *True* + *True* is well-typed in [33], in a program context where + is overloaded for integers and floating-point numbers but not for booleans.

In [26, 27], an overloading policy called *overloading by constructors* was proposed, defined by:

$\rho_{\mathtt{oc}}(A)$ holds if $A(o) = \{\forall\alpha_{i1}, \ldots, \alpha_{in_i}.\,\kappa_i.\,\tau_i\}^{i=1..n}$
implies $\Big(\tau_0 \leq \{\tau_i\}^{i=1..n}$, for some τ_0 s.t. $tv(\tau_0) = \{\alpha\}$
(for a single type variable α) and, for $i = 1, \ldots, n$:

1. $\kappa_i = \{o : \tau_0[\alpha_i'/\alpha] \mid \alpha_i' \in \{\alpha_{ij}\}^{j=1..n_i}\}$
2. $\tau_i = \tau_0[C_i\ \alpha_{i1} \cdots \alpha_{in_i}/\alpha]$, for some C_i, and
3. $C_i \neq C_j$, for all $j \in \{1, \ldots, n\}, j \neq i\ \Big)$

$\Gamma_{(==)}$ of Example 1 is a typing context according to $\rho_{\mathtt{oc}}$, but $\Gamma_{\mathtt{f}}$ of Example 6 and $\Gamma_{\mathtt{ins}}$ of Example 3 are not.

With $\rho_{\mathtt{oc}}$, *CS-SAT* has been shown in [32] to be solvable in polynomial time. This overloading policy is, however, very restrictive. It disallows, in particular, any constrained type with more than one element in the constraint set, like in:

Example 9. $\rho_{\mathtt{oc}}$ disallows the overloadings in:

$$\Gamma_* = \{\ \mathtt{+} : \mathit{Int} \to \mathit{Int} \to \mathit{Int},\quad \mathtt{+} : \mathit{Float} \to \mathit{Float} \to \mathit{Float},$$
$$\mathtt{*} : \mathit{Int} \to \mathit{Int} \to \mathit{Int},\quad \mathtt{*} : \mathit{Float} \to \mathit{Float} \to \mathit{Float},$$
$$\mathtt{*} : \forall\alpha.\,\{\mathtt{+} : \alpha \to \alpha \to \alpha, \mathtt{*} : \alpha \to \alpha \to \alpha\}.$$
$$\mathit{Matrix}\ \alpha \to \mathit{Matrix}\ \alpha \to \mathit{Matrix}\ \alpha\ \}$$

The (single parameter) overloading policy used in Haskell, less restrictive than $\rho_{\mathtt{oc}}$, can be defined as follows:

$\rho_{\mathtt{h}}(A)$ holds if $o \in \mathcal{O}$ implies that $A(o)$ is of the form
$\{\forall\alpha_{i_1}..\alpha_{i_{m_i}}.\kappa_i.(\tau[C_i\alpha_{i_1}..\alpha_{i_{m_i}}/\alpha])\}^{i=1..n}$ and, if $n > 1$:

1. $tv(\tau) = \alpha$, for a single type variable α
2. $C_i \neq C_j$, for $i \neq j$
3. $(o' : \tau') \in \kappa_i, 1 \leq i \leq n$, implies $tv(\tau') = \{\alpha'\} \subseteq \{\alpha_{i_1}..\alpha_{i_{m_i}}\}$

In [24], Helmut Seidl proved, using the type system of Tobias Nipkow and Christian Prehofer[21], that *CS-SAT* is EXPTIME complete for Haskell typing contexts. In [31], Dennis Volpano proved the same, using Geoffrey Smith's type system [26].

Example 9 was presented by Volpano [30] as a motivation for making a less restrictive proposal, called *parametric overloading*. Let us define also, for any set of types $\{\sigma_i\}^{i=1..n}$, where $\sigma_i = \forall\bar{\alpha}_i.\,\kappa_i.\,\tau_i$, that $lcg(\tau, \{\sigma_i\}^{i=1..n})$ holds if it holds that $lcg(\tau, \{\tau_i\}^{i=1..n})$. The overloading policy $\rho_{\mathtt{v}}$ in Volpano's parametric overloading can be defined inductively as follows:

1. $\rho_{\mathtt{v}}(\emptyset)$ holds;
2. $\rho_{\mathtt{v}}(A)$ holds if $A(x) = \{\forall\alpha_{i_1}.\ldots.\alpha_{i_{n_i}}.\,\kappa_i.\,\tau_i\}^{i=1..n}$, $n > 1$, implies:
   (a) $lcg(\tau, \{\tau_i\}^{i=1..n})$ holds, for some τ such that $tv(\tau) = \{\alpha\}$, for a single type variable α, and, for $i = 1, \ldots, n$, $\tau_i = \tau[C_i\ \alpha_{i_1} \ldots \alpha_{i_{n_i}}/\alpha]$;
   (b) for all $i, i' \in \{1, \ldots, n\}$, $C_i \neq C_{i'}$ if $i \neq i'$, and $(o : \tau') \in \kappa_i$ implies that $lcg(\tau', A(o))$ holds, for some τ' such that $tv(\tau') = \{\alpha'\}$, for a single type variable α'.
3. no mutual-recursion occurs in constraint-sets. More precisely, let o depend on o' in A if there exists $o : \forall\bar{\alpha}.\,\kappa.\,\tau$ in A such that $(o' : \tau') \in \kappa$, for some τ'. The condition of non-mutual recursion in constraint-sets requires the transitive closure of this dependency relation to be antisymmetric.

Volpano's parametric overloading is still restrictive. It disallows, for example, overloaded definitions whose types have a least common generalisation with more than one type variable. Volpano has shown in [30] that *CS-SAT* for parametric overloading is NP-hard.

System CT's overloading policy $\rho_{\mathtt{ct}}$ (Definition 8 below) relaxes restrictions imposed on types of overloaded symbols. Although *CS-SAT* is undecidable under system CT's overloading policy, we present in the next section an algorithm that is expected to decide *CS-SAT* in a vast majority of cases of practical interest. In cases where satisfiability cannot be decided, the algorithm terminates by using a configurable limit on the number of performed iterations.

Definition 6 (Well-formed typing context). *Let* $wfd(A)$ *hold if, for all* $(o : \forall\bar{\alpha}.\,\kappa.\,\tau) \in A$, *we have that* $A \models \kappa$ *is provable.*

DEFINITION 7 (NON-OVERLAPPING OVERLOADINGS).

$$nonOverlapping(A) = \Big(o \in \mathcal{O}, \{\sigma, \sigma'\} \subseteq A(o), \sigma' \neq \sigma, \\ \sigma = \forall \bar{\alpha}.\, \kappa.\, \tau, \sigma' = \forall \bar{\alpha}'.\, \kappa'.\, \tau', tv(\bar{\alpha}) \cap tv(\bar{\alpha}') = \emptyset \\ implies\ unify(\{\tau = \tau'\})\ fails\ \Big)$$

DEFINITION 8 (SYSTEM CT'S OVERLOADING POLICY). [6]

$$\rho_{\mathtt{ct}}(A) = \big(global(A)\ and\ nonOverlapping(A)\ and\ wfd(A)\big)$$

Example 10. Consider:[7]

A_1	$= \{\ o : Int,\quad o : Float,\quad o : \forall a.\, \{o : a\}.\, [a]\ \}$
A_2	$= \{\ o : Int,\quad o : Float,\quad o : \forall a.\, \{one : a\}.\, a\ \}$
A_3	$= \{\ o : Int,\quad o : Float,\quad o : \forall a.\, \{o : [a]\}.\, [a]\ \}$
A_4	$= \{\ o : Int,\quad o : Float,\quad o : \forall a.\, \{t : [[a]]\}.\, [a],$ $t : Int,\quad t : Float,\quad t : \forall a.\, \{o : a\}.\, [a]\ \}$
A_5	$= \{\ o : Int \to Int,\quad o : \forall a, b.\, \{o : a \to b\}.\, [a] \to b\ \}$
A_6	$= \{o : Int,\quad o : \forall a.\, \{o : a\}.\, [a]\ \}$

We have:

A_i	$\rho_{\mathtt{ct}}(A_i)$	Reason
A_1	true	$A_1 \models \{o : a\}$
A_2	false	$not\ nonOverlapping(A_2)$
A_3	false	$A_3 \not\models o : [a]$
A_4	false	$A_4 \not\models t : [[a]]$
A_5	false	$A_5 \not\models o : a \to b$
A_6	true	$A_6 \models o : a$

6. *CS-SAT*: A SOLUTION

Function *sat* (Figure 3) tests satisfiability of a constraint-set κ in a given typing context Γ. $sat(\kappa, \Gamma)$ either fails or gives a substitution S such that $\Gamma \models S\kappa$ is provable[8] and, furthermore, for any S' such that $\Gamma \models S'\kappa$ is provable, there exists a substitution R such that $S' = R \circ S$ (see Theorem 1 below). The substitution returned by *sat* is used, in system CT's type inference algorithm, to infer principal typings, by application of this substitution and constraint-set simplification (a process called *improvement* in e.g. [13, 15]).

[6]System CT could be modified to allow overlapping overloadings (by introducing a mechanism of choice between overlapping definitions), as in e.g. Haskell, but this has been left for future work and is not discussed in this paper.

[7]The basic idea for obtaining a proof of satisfiability of a constraint $o : \tau$ is to find a subset of assumptions of o in the relevant typing context with instances of type, say, $o : \forall \bar{\alpha}_i.\, \kappa_i.\, \tau_i$, for $i = 1, \ldots, n$, such that the least common generalisation of $\{\tau_i\}^{i=1..n}$ yields τ and each κ_i is itself satisfiable in this typing context.

[8]We do not require instead that $S\Gamma \models S\kappa$ holds because we are considering only global overloading in this paper.

The test of satisfiability of a constraint-set $\kappa = \{o_i : \tau_i\}^{i=1..n}$ in a typing context Γ requires determining, firstly, the largest set of assumptions $\{o_i : \forall \bar{\alpha}_i.\kappa_i.\, \tau_i'\}^{i=1..n}$ in Γ such that there exists a substitution that unifies each τ_i with τ_i' . We call this the *sat-set* of κ in Γ. It is the first component given by function *satset*, defined in Figure 2 (the second component is the most general unifying substitution).

Secondly, for κ to be satisfiable in Γ, there must exist, for each $o_i : \tau_i \in \kappa$, $i = 1, \ldots, n$, at least one $(o_i : \kappa'.\, \tau')$ in the sat-set of κ in Γ such that κ' is itself satisfiable in Γ. For example, $\kappa = \{\mathtt{(==)} : [C] \to [C] \to Bool\}$ is not satisfiable in $\Gamma_{\mathtt{ins}}$ (Example 3), for any parameterless type constructor C distinct from *Int* and *Float*, because $\{\mathtt{(==)} : C \to C \to Bool\}$ is itself not satisfiable in $\Gamma_{\mathtt{ins}}$.

$sat(\kappa, \Gamma)$ is given by $sats(\kappa, \Gamma, \varkappa, \emptyset)$, where $\varkappa$ is a positive integer constant, chosen as a limit on the number of recursive calls for which satisfiability cannot be otherwise decided (as explained below).[9] The last parameter in $sats(\kappa, \Gamma, \varkappa, \overline{\overline{\kappa'}})$, initially the empty set, contains generalizations of constraints which have already been tested for satisfiability (in a particular branch of the tree of recursive calls to *sats*).

A call to $sats\big(\{o_i : \tau_i\}^{i=1..n}, \Gamma, \varkappa, \overline{\overline{\kappa}}\big)$ such that $satset(\{(o_i : \tau_i, \Gamma_i)\}^{i=1..n} = \{(\{o_i : \kappa_{ij}.\, \tau_{ij}\}^{i=1..n}, S_j)\}^{j=1..m}$ originates m (possibly zero) recursive calls to *sats*, one for each value of $j \in \{1, \ldots, m\}$. *sats* tests whether there exist, in these recursive calls, so-called downward and, if not, looping occurrences of constraints, as explained by means of an example below. If there is a looping occurrence in some constraint-set κ_j, *sats* calls itself (to test satisfiability of this looping constraint) on a proper subset of the original typing context Γ. This is also explained in the following.

Relations defining downward and looping occurrences are defined in Figure 4. Additional notation used in the definition of *sats* is given below:

$$\overline{\overline{\kappa_1}} \oplus \kappa_2 = \overline{\overline{\kappa_1}} \cup \overline{\overline{\kappa_2}}$$
$$\Gamma \ominus \kappa = \Gamma - \overline{\overline{\kappa}}$$

Example 11. Consider assumption set $A_6 = \{o : Int,\ o : \forall a.\, \{o : a\}.\, [a]\}$ of Example 10, and $sats(\{o : a\}, A_6, \varkappa, \emptyset)$. We have that

$$satset(\{o : a\}, A_6) = \{\ (\{o : Int\}, \{a \mapsto Int\}), \\ (\{o : \{o : a'\}.[a']\}, \{a \mapsto [a']\})\ \}$$

where a' is a fresh type variable. Constraint-set $\{o : a'\}$ must then be tested for satisfiability, by a recursive call to *sats*.

Since $\{o : a'\} \lessdot \emptyset$, the recursive call to *sats* is given by:

[9]Formally, $\varkappa$ should be an additional parameter in the definition of *sat*, but we allow ourselves a bit of informality here.

$$
\begin{array}{l}
\mathit{satset}(\emptyset) \qquad\qquad\quad = \{(\emptyset, \mathit{id})\} \\
\mathit{satset}(\{o:\tau\},\Gamma) \quad = \{(\{o:\kappa.\,\tau'\}, S) \mid o:\forall\bar\alpha.\,\kappa.\,\tau' \in \Gamma,\ \bar\alpha \cap tv(\tau) = \emptyset, \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{unify}(\{\tau = \tau'\}) = S\} \\
\mathit{satset}(\{o:\tau\} \cup \kappa, \Gamma) = \\
\quad \textbf{let}\ \ \{(\{o:\kappa_i.\,\tau_i\}, S_i)\}^{i=1..n} = \mathit{satset}(\{o:\tau\},\Gamma) \\
\qquad\quad \{(\Gamma_{ij}, S_{ij})\}^{j=1..m_i} \qquad = \mathit{satset}(\{o:S_i\tau \mid o:\tau\in\kappa\},\Gamma),\ \text{for } i = 1..n \\
\quad \textbf{in}\ \ \{(\Gamma_{ij} \cup \{o:\kappa_i.\,\tau_i\}, S_{ij} \circ S_i)\}^{i=1..n, j=1..m_i}
\end{array}
$$

Figure 2: *satset*

$$
\begin{array}{l}
\mathit{sat}\,(\kappa,\Gamma) = \mathit{sats}\big(\{\kappa,\Gamma,\varkappa,\emptyset\big) \\
\\
\mathit{sats}(\emptyset,\Gamma,\varkappa,\overline{\overline{\kappa}}) = \mathit{id} \\
\mathit{sats}\,\big(\{o_i:\tau_i\}^{i=1..n},\Gamma,\varkappa,\overline{\overline{\kappa}}\big) = \\
\ \textbf{let}\ \ \big\{\big(\{o_i:\kappa_{ij}.\,\tau_{ij}\}^{i=1..n}, S_j\big)\big\}^{j=1..m} = \mathit{satset}\big(\{(o_i:\tau_i)\}^{i=1..n},\Gamma\big) \\
\ \textbf{in}\ \ \textbf{if}\ \ m = 0\ \ \textbf{then}\ \ \text{fail}\ \ \textbf{else} \\
\qquad \textbf{let}\ \ \text{for } j = 1,\ldots,m\text{:} \\
\qquad\qquad \kappa_j = \bigcup_{i=1,\ldots,n} S_j\kappa_{ij}, \\
\qquad\qquad (\Gamma_j,\varkappa_j) = \\
\qquad\qquad\qquad \textbf{if}\ \ \kappa_j \lessdot \overline{\overline{\kappa}}\ \ \texttt{(downward occurrence)}\ \ \textbf{then}\ \ (\Gamma,\varkappa)\ \ \textbf{else} \\
\qquad\qquad\qquad \textbf{if}\ \overline{\overline{\kappa}} \le \kappa_j\ \ \texttt{(looping occurrence)}\ \ \textbf{then}\ \ (\Gamma \ominus \{o_i:\kappa_{ij}.\,\tau_{ij}\}^{i=1..n},\varkappa)\ \ \textbf{else} \\
\qquad\qquad\qquad \textbf{if}\ \ \varkappa > 0\ \ \textbf{then}\ \ (\Gamma,\varkappa-1)\ \ \textbf{else}\ \ \text{fail} \\
\qquad\quad \overline{\overline{\kappa_0}} = \overline{\overline{\kappa}} \oplus \{o_i:\tau_i\}^{i=1..n} \\
\qquad\quad \mathbb{S} = \big\{S'_j \circ S_j \mid S'_j = \mathit{sats}(\kappa_j,\Gamma_j,\varkappa_j,\overline{\overline{\kappa_0}})\big\}^{j=1..m} \\
\qquad \textbf{in}\ \ \textbf{if}\ \ \mathbb{S} = \emptyset\ \ \textbf{then}\ \ \text{fail}\ \ \textbf{else}\ \ \bigcap\mathbb{S}
\end{array}
$$

Figure 3: *sat*

$$
\begin{array}{l}
\kappa' \lessdot \overline{\overline{\kappa}} = \big(\exists (o:\tau)\in\kappa' \text{ such that } \tau \lessdot \kappa(o)\big) \\
\\
\tau \lessdot \emptyset \text{ holds}, \quad \tau \lessdot \{\tau'\} = \big(\tau \nsim \tau' \text{ or } \tau < \tau'\big), \quad \tau \lessdot \mathbb{T} = \big(\tau \lessdot \{\tau'\},\ \text{for some } \tau' \in \mathbb{T}\big) \\
\\
(C\,\tau_1 \ldots \tau_n) \nsim (C'\,\tau'_1 \ldots \tau'_m) = \ \big(C \neq C' \text{ or } \tau_i \nsim \tau'_i, \text{for some } i \in \{1,\ldots,n\}\big) \\
\tau \nsim \tau' \text{ does not hold, otherwise} \\
\\
\overline{\overline{\kappa}} \le \kappa' = \big(\exists (o:\tau)\,\bar\in\,\overline{\overline{\kappa}} \text{ such that } \tau \le \kappa(o)\big) \quad (o:\tau)\,\bar\in\,\overline{\overline{\kappa}} = (o:\overline{\overline{\tau}} \in \overline{\overline{\kappa}}) \\
\\
\tau \le \mathbb{T} = \big(\exists\ \tau' \in \mathbb{T} \text{ such that } \tau \le \tau'\big)
\end{array}
$$

Figure 4: Relations on constraints and types

$$sats(\{o : a'\}, A_6, \varkappa, \{\!|o : a|\!\})$$

where $\{o : a\}$ is marked to have already been tested (and $\{\!|o : a|\!\} = \forall a.\, \{o : a\}.\, a$). $satset(\{o : a'\}, A_6)$ is analogous. But now, in the recursive call to *sats*, we have (no downward but) a looping occurrence, given by $\{\!|o : a|\!\} \leq \{o : a'\}$. The next recursive call becomes then:

$$sats(\{o : a''\}, A_6', \varkappa, \{\!|o : a|\!\})$$

where $A_6' = A_6 - \{\forall a.\, \{o : a\}.\, [a]\}$.

We have that $satset(o : a'', A_6') = \{(o : Int, \{a'' \mapsto Int)\}$ and thus that $sats(\{o : a''\}, A_6', \varkappa, \{\!|o : a|\!\}) = \{a'' \mapsto Int\}$. Using this, we have, in the call to $sats(\{o : a'\}, A_6, \varkappa, \{\!|o : a|\!\})$, that

$$\mathbb{S} = \{\{a' \mapsto Int,\ a' \mapsto [Int]\}\}$$

This gives $\cap\mathbb{S} = \{a' \mapsto \alpha\}$, where α is a fresh type variable. Thus, in the call to $sats(\{o : a\}, A_6, \varkappa, \emptyset)$, we have that

$$\mathbb{S} = \{\{a \mapsto Int,\ \{a \mapsto [\alpha]\}\}$$

giving $\cap\mathbb{S} = \{a \mapsto \alpha'\}$, for some fresh type variable α'.

In situations where there are no no downward nor looping occurrences, a predefined limit on the number of recursive calls to *sats* is decremented, until zero, thus guaranteeing termination. This is illustrated in the following.[10]

Example 12. Let $\Gamma = \{\ o : Int \to Bool,\ o : Char \to Int,\ o : \forall a, b.\, \{o : a \to b\}.\, T^2\ a \to b\}$

where $T^i\ a$ is used as abbreviation for the simple type given by $T\ (T\ \ldots (T\ a))$ with i occurrences of T. Let $\kappa = \{o : \alpha \to T\ \alpha\}$. Then $sat(\kappa, \Gamma)$ loops forever if a limit $\varkappa$ is not used (the limit is placed on the number of recursive calls to *sat* for which the parameters yield neither looping nor downward occurrences):

$$\begin{array}{l} sat(\kappa, \Gamma) \\ = sats(\kappa, \Gamma, \varkappa, \emptyset)\}) \\ = sats(\{(o : a_1 \to T^3 a_1, \Gamma, \varkappa - 1, \overline{\overline{\kappa}})\}) \circ S_1 \\ \quad \text{where } S_1 = \{\alpha \mapsto T^2 a_1, b_1 \mapsto T^3 a_1\} \\ = sats(\{(o : a_2 \to T^5 a_2, \Gamma, \varkappa - 2, \overline{\overline{\kappa}} \cup \{\!|o : a_1 \to T^2 a_1|\!\}\}) \\ \qquad\qquad \circ S_2 \circ S_1 \\ \quad \text{where } S_2 = \{a_1 \mapsto T^2 a_2, b_2 \mapsto T^5 a_2\} \\ = \ldots \text{ (loops if test on } \varkappa \text{ is not used)} \end{array}$$

[10]The example was presented to us by Martin Sulzmann.

However, for all cases that would cause *sats* to loop forever if a test on $\varkappa$ was not used, a type error will be correctly reported, since the constraint-set is not satisfiable (see theorem 3). On the other hand, with the introduction of the test, there can exist satisfiable constraints for which the test on $\varkappa$ incorrectly reports failure, as demonstrated in the following.

Example 13. We consider an instance of the Post Correspondence Problem (PCP) presented in [32]. A similar one appears in [26], but involves overlapping assumptions, and the given set of assumptions is thus not considered (by *nonOverlapping*) as a valid typing context (according to $\rho_{\mathtt{ct}}$). Typing context $\Gamma_{\mathtt{p}}$ and constraint-set $\kappa_{\mathtt{p}}$ given below are such that $\kappa_{\mathtt{p}}$ is satisfiable in $\Gamma_{\mathtt{p}}$ if and only if its solution to the corresponding instance of PCP (constructed as given in [32]) exists. Typing context $\Gamma_{\mathtt{p}}$ is as follows:

$$\begin{array}{ll} \Gamma_{\mathtt{p}} = \{ & \\ p: & (C_1 \to C_0 \to C) \to (C_1 \to C_0 \to C_1 \to C) \to C_1' \\ p: & (C_0 \to C_1 \to C_1 \to C) \to (C_1 \to C_1 \to C) \to C_2' \\ p: & (C_1 \to C_0 \to C_1 \to C) \to (C_0 \to C_1 \to C) \to C_3' \\ p: & \forall a, b.\, \{p : a \to b \to c\}. \\ & (C_1 \to C_0 \to a) \to (C_1 \to C_0 \to C_1 \to b) \to (C_1' \to c) \\ p: & \forall a, b, c.\, \{p : a \to b \to c\}. \\ & (C_0 \to C_1 \to C_1 \to a) \to (C_1 \to C_1 \to b) \to (C_2' \to c) \\ p: & \forall a, b.\, \{p : a \to b \to c\}. \\ & (C_1 \to C_0 \to C_1 \to a) \to (C_0 \to C_1 \to b) \to (C_3' \to c)\ \} \end{array}$$

where C,C_0,C_1,C_1' and C_2' are unary type constructors.

Let $\kappa_{\mathtt{p}} = \{p : a \to a \to b\}$. We have that $sat(\kappa_{\mathtt{p}}, \Gamma_{\mathtt{p}})$ fails if, but only if, $\varkappa < 1$, since:

$$satset(\{(p : a \to a \to b, \Gamma_{\mathtt{p}})\}) = \{(\{p : \kappa.\, \tau\}, S)\}$$

where

$$\begin{array}{l} \kappa.\, \tau = \{p : a_1 \to b_1 \to c_1\}.\, (C_1 \to C_0 \to a_1) \to \\ \qquad (C_1 \to C_0 \to C_1 \to b_1) \to (C_1' \to c_1)\} \\ S = \{\{(\ a_1 \mapsto C_1 \to b_1, b \mapsto (C_1' \to c_1), \\ \qquad a \mapsto (C_1 \to C_0 \to C_1 \to b_1))\}\} \end{array}$$

and the recursive call to *sats* is

$$sats(\{(p : (C_1 \to b_1) \to b_1 \to c_1, \Gamma_p, \varkappa{-}1, \{\!|p : a \to a \to b|\!\})\})$$

This involves the computation of

$$satset(\{(p : (C_1 \to b_1) \to b_1 \to c_1, \Gamma_{\mathtt{p}})\})$$

which gives

$$\{\ (p : (C_1 \to C_0 \to C_1 \to C) \to (C_0 \to C_1 \to C) \to C_3', S_1),\\ (p : \{p : a_2 \to b_2 \to c_2\}.\ (C_1 \to C_0 \to C_1 \to a_2) \to\\ (C_0 \to C_1 \to b_2) \to (C_3' \to c_2), S_2)\ \}$$

where: $S_1 = id \dagger \{(b_1 \mapsto C_0 \to C_1 \to C, c_1 \mapsto C_3')\}$ and $S_2 = id \dagger \{(a_2 \mapsto b_2, b_1 \mapsto C_0 \to C_1 \to b_2, c_1 \mapsto (C_3' \to c_2))\}$.

The first of these results causes a recursive call to *sats* with an empty constraint-set, which yields the identity substitution. The second involves a recursive call to *sats* given by:

$$sats\Big(\Big\{\big(p : b_2 \to b_2 \to c_2,\ \Gamma_{\mathrm{p}} - \{p : \forall a, b.\, \{p : a \to b \to c\}.\\ (C_1 \to C_0 \to C_1 \to a) \to (C_0 \to C_1 \to b) \to (C_3' \to c)\},\\ \varkappa - 1, \{\!\|p : a \to a \to b,\ p : (C_1 \to b_1) \to b_1 \to c_1\|\!\},\ \big)\Big\}\Big)$$

This also succeeds. The final result is given by

$$\bigcap\{S \circ S_1 \circ id, S \circ S_2 \circ S_3 \circ id\}$$

where $S_3 = \{a_3 \mapsto C_1 \to b_3,\ b_2 \mapsto C_1 \to C_0 \to C_1 \to b_3,\ c_2 \mapsto C_1' \to c_3\}$.

For any value ℓ, we can find a satisfiable constraint κ_ℓ such that $sat(\kappa_\ell, \Gamma_{\mathrm{p}})$ fails with $\varkappa = \ell$. For example, if $\ell = \varkappa = 1$, we can take $\kappa_\ell = \{p : (C_0 \to C_1 \to a) \to (C_1 \to C_1 \to a) \to b)$; if $\ell = \varkappa = 2$, we can take $\kappa_\ell = \{p : (C_0 \to C_1 \to C_1 \to C_0 \to C_1 \to a) \to (C_1 \to C_1 \to C_1 \to C_1 \to a) \to b)$, and so on.

Function *sat* satisfies the following (proofs can be found in [4]):

THEOREM 1 (CORRECTNESS OF *sat*). *For any set of assumptions* Γ *such that* $\rho_{\mathsf{ct}}(\Gamma)$ *holds and any constraint-set* κ*, if* $sat\,(\kappa, \Gamma) = S$*, for some* S*, then* $\Gamma \models S\kappa$ *is provable and, furthermore, for any* S' *such that* $\Gamma \models S'\kappa$ *is provable, there exists* R *such that* $S' = R \circ S$.

THEOREM 2 (*sat* ALWAYS TERMINATES). *For any set of assumptions* Γ *such that* $\rho_{\mathsf{ct}}(\Gamma)$ *holds and any constraint-set* κ*, either* $sat(\kappa, \Gamma)$ *fails or* $sat(\kappa, \Gamma) = S$*, for some* S.

Although *sat* is not complete (with respect to *CS-SAT*), that is, although failure of $sat\,(\kappa, \Gamma)$ does not imply that there exists no S such that $\Gamma \models S\kappa$ is provable (see Example 13), we have:

THEOREM 3. *For any set of assumptions* Γ *such that* $\rho_{\mathsf{ct}}(\Gamma)$ *holds and any constraint-set* κ*, if* $sat(\kappa, \Gamma)$ *fails with* $\varkappa = \ell$ *(for some* ℓ*) and there exists* S *such that* $\Gamma \models S\kappa$ *is provable, then there exist* $\ell' > \ell$ *and* S' *such that* $sat(\kappa, \Gamma) = S'$ *if* $\varkappa = \ell'$*, and* $S' = R \circ S$*, for some* R.

$$\begin{array}{l}\hat{\pi}(\emptyset) = \emptyset\\ \hat{\pi}(\{o : \tau\} \cup \kappa) = \textbf{let}\ \ \mathbb{K} = \hat{\pi}(\kappa)\ \textbf{in}\\ \quad \textbf{if}\ \ \exists\, \kappa' \in \mathbb{K} \text{ such that } tv(\tau) \cap tv(\kappa') \neq \emptyset\\ \qquad \textbf{then}\ \ \mathbb{K} - \{\kappa'\} \cup \{\kappa' \cup \{o : \tau\}\}\\ \qquad \textbf{else}\ \ \mathbb{K} \cup \{\{o : \tau\}\}\end{array}$$

Figure 5: Constraint-set projections

$$\begin{array}{l}sat(\kappa, \Gamma) = sat'(\kappa, \Gamma, \varkappa, \emptyset)\\ \\ sat'(\kappa, \Gamma, \varkappa, \overline{\overline{\kappa_0}}) = S_1 \circ S_2 \circ \cdots S_n\\ \quad \text{where}\ \ \{\kappa_i\}^{i=1..n} = \hat{\pi}(\kappa)\\ \qquad\quad S_i = sats(\kappa_i, \Gamma, \varkappa, \overline{\overline{\kappa_0}})\end{array}$$

Figure 6: Satisfiability of projections

7. OPTIMIZATION

The verification of satisfiability of a constraint-set κ in a typing context Γ can be "optimized" by noting that constraints with types that have no type variables in common can be tested separately. Figure 5 defines so-called *projections* of constraint-set κ, which are the subsets of constraints in κ whose types have at least one common type variable. A projection can be seen as an automatically derived set of "functionally dependent" constraints. Figure 6 contains a definition of $sat(\kappa, \Gamma)$ based on the composition of substitutions given by computing *sat* separately for each projection of κ. The definition of *sats* is changed to call sat' instead of making a direct recursive call to *sats*.

It is not difficult to see that the size of the sat-set of a constraint-set $\{o_i : \tau_i\}^{i=1..n}$ is, in the worst case, $s_1 \times \cdots \times s_n$, where s_i is the size of the sat-set of the constraint $o_i : \tau_i$. Worst cases occur when each constraint represents an independent projection. For a simple example, consider:

$$\Gamma = \{o_1 : \mathit{Int},\ o_1 : \mathit{Bool},\ o_1 : \mathit{Char},\ o_2 : \mathit{Int},\ o_2 : \mathit{Float}\}$$

Then $satset(\{o_1 : a,\ o_2 : b\}, \Gamma)$ returns a sat-set that is a combination of all elements in the sat-set given by $satset(\{o_1 : a\}, \Gamma) = \{o_1 : \mathit{Int},\ o_1 : \mathit{Bool},\ o_1 : \mathit{Char}\}$ with all elements in the sat-set given by $satset(\{o_2 : a\}, \Gamma) = \{o_2 : \mathit{Int},\ o_2 : \mathit{Float}\}$. This combination is fruitless in the case of independent projections like $\{o_1 : a\}$ and $\{o_2 : b\}$, and is avoided by computing $sats(\{o_1 : a\}, \Gamma)$ and $sats(\{o_2 : b\}, \Gamma)$ separately, and just composing the obtained substitutions.

8. CONCLUSION

This article has addressed the problem of constraint-set satisfiability, in the presence of context-dependent overloading and parametric polymorphism. An algorithm is presented that uses a finite sequence of unifications to perform constraint-set satisfiability, and does not require the use of a restrictive overloading policy. The article has also reviewed

some related works on constraint-set satisfiability and overloading policies.

Related recent works that seek a foundation for the support of constrained polymorphism include Martin Sulzmann's [28], based on the explicit use of *constraint handling rules*, and Bart Demoen, María de la Banda and Peter Stuckey's [6], based on the transformation of constraint-set satisfiability into resolution in constraint logic programs.

A prototype of a type inference algorithm is available at `http://www.dcc.ufmg.br/~damiani/CT/CT.zip`. The examples presented in this paper (and many others) have been tested with this prototype and are included in this file. The prototype includes a parser (based on Parsec's monadic parser combinators [12]) for a language that is basically core-Haskell without type classes and with support for overloading as described in this paper. The implementation is an adaptation of a type inference algorithm written by Mark Jones [16] and, essentially, it performs type inference allowing mutually dependent definitions and polymorphic recursion [29, 11], followed by constraint-set satisfiability, using the algorithm described in this paper, improvement (application of the substitution returned by *sat* followed by constraint-set simplification) and ambiguity checking.

9. REFERENCES

[1] Carlos Camarão and Lucília Figueiredo. Type Inference for Overloading without Restrictions, Declarations or Annotations. In *Proceedings of the 4th Fuji International Symposium on Functional and Logic Programming* (FLOPS'99), number 1722 in Lecture Notes in Computer Science, pages 37–52, 1999.

[2] Kung Chen, Paul Hudak, and Martin Odersky. Parametric Type Classes. In *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, pages 170–181, 1992.

[3] Luís Damas and Robin Milner. Principal type schemes for functional programs. In *Proceedings of the 9th ACM Symposium on Principles of Programming Languages* (POPL'82), pages 207–212, 1982.

[4] Lucília Figueiredo, Carlos Camarão and Cristiano Vasconcelos. Constraint-set Satisfiability for Overloading. Technical report, UFMG, 2003.

[5] Fergus Henderson, David Jeffery and Zoltan Zomogyi. Type Classes in Mercury. Technical Report 98/13, University of Melbourne, 1998.

[6] Bart Demoen, María García de la Banda, and Peter J. Stuckey. Type Constraint Solving for Parametric and Ad-hoc Polymorphism. In *Proceedings of the 22nd Australasian Computer Science Conference*, 1999.

[7] Dominic Duggan, Gordon Cormack, and John Ophel. Kinded type inference for parametric overloading. *Acta Informatica*, 33(1):21–68, 1996.

[8] Dominic Duggan and John Ophel. Open and closed scopes for constrained genericity. *Theoretical Computer Science*, 275(1–2):215–258, 2002.

[9] Dominic Duggan and John Ophel. Type checking multi-parameter type classes. *Journal of Functional Programming*, 12(2):135–158, 2002.

[10] Cordelia Hall, Kevin Hammond, Simon Peyton Jones, and Philip Wadler. Type Classes in Haskell. *ACM Transactions on Programming Languages and Systems*, 18(2):109–138, 1996.

[11] Fritz Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2), 253–289, 1993.

[12] Daan Leijen and Erik Meijer. Parsec: Direct Style Monadic Parser Combinators for the Real World. Technical Report UU-CS-2001-35, Department of Computer Science, Universiteit Utrecht, 2001. `http://www.cs.uu.nl/ daan/parsec.html`.

[13] Mark Jones. *Qualified Types*. Cambridge University Press, 1994.

[14] Mark Jones. A system of constructor classes: overloading and higher-order polymorphism. *Journal of Functional Programming*, 5(1):1–36, 1995.

[15] Mark Jones. Simplifying and Improving Qualified Types. In *Proceedings of the 1st ACM Conference on Functional Programming and Computer Architecture* (FPCA'95), pages 160–169, 1995.

[16] Mark Jones. Typing Haskell in Haskell. In *Proceedings of the 1999 Haskell Workshop*. Published in Technical Report UU-CS-1999-28, Department of Computer Science, University of Utrecht. Available at `http://www.cse.ogi.edu/~mpj/thih/`

[17] Stefan Kaes. Parametric overloading in polymorphic programming languages. In *Proceedings of the 2nd European Symposium on Programming* (ESOP'88), number 300 in Lecture Notes in Computer Science, pages 131–144, 1988.

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

[19] John Mitchell. Polymorphic type inference and containment. *Information and Computation*, 76(2/3):211–249, 1988.

[20] John Mitchell. *Foundations for Programming Languages*. MIT Press, 1996.

[21] Tobias Nipkow and Christian Prehofer. Type Reconstruction for Type Classes. *Journal of Functional Programming*, 1(1):1–100, 1993.

[22] Martin Odersky, Philip Wadler, and Martin Wehr. A Second Look at Overloading. In *Proceedings of the 7th ACM Conference on Functional Programming and Computer Architecture*, pages 135–146, 1995.

[23] J.A. Robinson. A machine oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

[24] Helmut Seidl. Haskell Overloading is *DEXPTIME* complete. *Information Processing Letters*, 52(2):57–60, 1994.

[25] Mark Shields and Simon Peyton Jones. Object-oriented style overloading for Haskell. In *Workshop on Multi-Language Infrastructure and Interoperability* (BABEL'01), 2001. Available at `http://haskell.readscheme.org/lang_sem.html`.

[26] Geoffrey Smith. *Polymorphic Type Inference for Languages with Overloading and Subtyping*. PhD thesis, Cornell University, 1991.

[27] Geoffrey Smith. Principal type schemes for functional programs with overloading and subtyping. *Science of Computer Programming*, 23(2-3):197–226, 1994.

[28] Peter Stuckey and Martin Sulzmann. A Theory of Overloading. In *Proceedings of the 7$^{\text{th}}$ ACM International Conference on Functional Programming*, pages 167–178, 2002.

[29] Cristiano Vasconcelos, Lucília Figueiredo, and Carlos Camarão. A Practical Type Inference for Polymorphic Recursion using Haskell. *Journal of Universal Computer Science*, 9(8):873–890, 2003.

[30] Dennis Volpano. Haskell-style Overloading is NP-hard. In *Proceedings of the 1994 International Conference on Computer Languages*, pages 88–95, 1994.

[31] Dennis Volpano. Lower Bounds on Type Checking Overloading. *Information Processing Letters*, 57(1):9–14, 1996.

[32] Dennis Volpano and Geoffrey Smith. On the Complexity of ML Typability with Overloading. In *Proceedings of the ACM Symposium on Functional Programming and Computer Architecture.*, number 523 in Lecture Notes in Computer Science, pages 15–28, 1991.

[33] Philip Wadler and Stephen Blott. How to make *ad-hoc* polymorphism less *ad hoc*. In *Proceedings of the 16$^{\text{th}}$ ACM Symposium on Principles of Programming Languages* (POPL'89), pages 60–76. ACM Press, 1989.