Journal of Brazilian Computer Society manuscript No.
(will be inserted by the editor)

Terminating Constraint Set Satisfiability and Simplification Algorithms for Context-Dependent Overloading

Rodrigo Ribeiro · Carlos Camarão · Lucília Figueiredo

Received: date / Accepted: date

Abstract Algorithms for constraint set satisfiability and simplification of Haskell type class constraints are used during type inference in order to allow the inference of more accurate types and to detect ambiguity. Unfortunately, both constraint set satisfiability and simplification are in general undecidable, and the use of these algorithms may cause non-termination of type inference. This paper presents algorithms for these problems that terminate on any given input, based on the use of a criterion that is tested on each recursive step.

The use of this criterion eliminates the need of imposing syntactic conditions on Haskell type class and instance declarations in order to guarantee termination of type inference in the presence of multi-parameter type classes, and allows program compilation without the need of compiler flags for lifting such restrictions. Undecidability of the problems implies the existence of instances for which the algorithm incorrectly reports unsatisfiability, but we are not aware of any practical example where this occurs.

Keywords First keyword · Second keyword · More

1 Introduction

Haskell's type class system [18,5] extends the Hindley-Milner type system [16] with constrained polymorphic types, in order to support overloading. Type class constraints may occur in types of expressions involving overloaded names (or symbols), and restrict the set of types to which quantified type variables may be instantiated, to those types for which these type constraints are satisfied, according to types of definitions that exist in a relevant context.

A type class declaration specifies the name and parameters of the class, and the principal type of names which can then be overloaded in instance definitions. For example:

```
class Eq a where
  (==):: a → a → Bool
  (/=):: a → a → Bool
```

is a declaration of type class Eq, with parameter a, that specifies the principal types of `(==)` and `(/=)`. Function `(==)` has type $\forall\, a.\, Eq\ a \Rightarrow a \rightarrow a \rightarrow Bool$, where constraint $Eq\ a$ indicates that type variable a cannot be instantiated to an arbitrary type, but only to a type that has been defined as an instance of class Eq.

An instance of a type class specifies instance types for type class parameters, and gives definitions of the overloaded names specified in the class. The type of each overloaded name in an instance definition is obtained by substituting type class parameters with corresponding instance types. For example, the following instance declarations specify definitions of the equality operator for types Int and for polymorphic lists, respectively:

Rodrigo Ribeiro, Carlos Camarão
Instituto de Ciências Exatas, Departamento de Ciência da Computação, Universidade Federal de Minas Gerais. E-mail: {camarao,rribeiro}@dcc.ufmg.br.

Lucília Figueiredo
Instituto de Ciências Exatas e Biológicas, Departamento de Computação, Universidade Federal de Ouro Preto. E-mail: lucilia@iceb.ufop.br

```
instance Eq Int where
  (==) = primEqInt

instance Eq a ⇒ Eq [a] where
  [ ]     == [ ]     = True
  (a:x) == (b:y) = a == b && x == y
  _       == _       = False
```

For a base type, like *Int*, a corresponding predefined operation is provided. The definition of equality for lists of elements of an arbitrary type uses the equality test for elements of this type. Constraint *Eq a* must be specified as the *context* for the *head* *Eq* `[a]` of the instance declaration. A *context* is a set of type class constraints, and constraint π is the *head* of a qualified constraint $P \Rightarrow \pi$, where P is a set of type class constraints.

As an aside, type classes in Haskell may also contain default definitions of the overloaded names, in order to avoid repeating the same definitions in instances.

Class constraints introduced on the types of overloaded symbols occur also on the types of expressions defined in terms of these symbols. For example, consider the following function that tests list membership:

```
elem a []     = False
elem a (b:x) = a == b || elem a x
```

The principal type of *elem* is $\forall a.\, Eq\, a \Rightarrow a \rightarrow [a] \rightarrow Bool$. Constraint *Eq a* occurs in the type of *elem* due to the use of the equality operator `(==)` in its definition.

Haskell restricts type classes to have a single parameter but the extension to multi-parameter type classes, called Haskell+mptcs in the sequel, is widely used.

Type inference for constrained type systems rely on constraint set simplification, which, for the case of type classes, essentially amounts to performing (so-called) *context reduction*. Constraint set simplification yields equivalent constraint sets, and are useful for providing simpler types for expressions. Context reduction simplifies constraints by substituting constraints or removing resolved constraints according to available instance definitions, besides removing duplicate constraints or substituting constraints according to the class hierarchy.

As an example, context *Eq*`[t]` is reduced to *Eq t*, for any type t, in the presence of instance *Eq*`[a]` with context *Eq a*.

Improvement [13] is also a process of simplification of constrained types, but it is of a different nature, and is used in type inference to avoid ambiguity and to infer more informative types. Improvement is fundamentally based on constraint set satisfiability: it is a process of transforming a constraint set P into a constraint set obtained by applying a substitution S to P so that the set of satisfiable instances of P is preserved.

The mechanism of functional dependencies and other alternatives have been proposed to deal with improvement [14,7,11,10,4], for detection of ambiguity and for specialization of constrained types in the presence of multi-parameter type classes. We do not discuss improvement specifically in this paper, but focus on constraint set satisfiability, which is only used for the implementation of improvement or any alternative approach.

Unfortunately, both constraint set satisfiability and simplification are in general undecidable problems [6], and the use of computable functions for solving these problems may cause non-termination of type inference.

This paper presents algorithms for constraint set satisfiability and simplification that use a termination criterion which is based on a measure of the sizes of types in type constraints. The sequence of constraints that unify with a constraint axiom in recursive calls of the function that checks satisfiability or simplification of a type constraint is such that either the sizes of types of each constraint in this sequence is decreasing or there exists at least one type parameter position with decreasing size.

The use of this criterion eliminates the need for imposing syntactic conditions on Haskell type class and instance declarations in order to guarantee termination of type inference in the presence of multi-parameter type classes, and allows program compilation without the need of compiler flags for lifting such restrictions.

The use of a termination criterion implies that there exist well-typed programs for which the presented algorithm incorrectly reports unsatisfiability. However, practical examples where this occurs are expected to be very rare. The algorithms have been implemented and tested by using a prototype front-end for Haskell, available at the mptc github repository. The algorithm works as expected when subjected to examples mentioned in the literature, Haskell libraries that use multi-parameter type classes and many tests, including those used by the mostly used Haskell compiler[19], GHC, involving all pertinent GHC extensions.

Restrictions imposed on class and instance declarations in Haskell, in Haskell+mptcs and in GHC, and GHC compilation flags used to avoid these restrictions [20], are summarized in Section 2. Section 3 reviews entailment and satisfiability relations on type class constraints. Section 4 gives a definition of a computable function that returns the set of satisfiable substitutions of a given constraint set P, when it terminates. Subsection 4.1 defines a termination criterion and redefines this computable function in order to use this criterion.

Section 5 defines a constraint set simplification computable function, based on the same termination criterion. Section 6 concludes.

2 Restrictions over Class and Instance Declarations

This section summarizes the restrictions imposed on class and instance declarations in Haskell, Haskell+mptcs and in GHC, and GHC compilation flags used to avoid these restrictions.

By default, GHC follows the Haskell language specification (i.e. the Haskell 98 report [8]), which imposes the following restrictions.

1. Each class declaration must have exactly one parameter.
2. The head of a qualified constraint in an instance declaration must have the form $C(T\,\overline{\alpha})$, where C denotes a class name, T a type constructor and $\overline{\alpha}$ a sequence of distinct type variables. Such overbar notation is used extensively in this paper: $\overline{x}$ denotes a possibly empty sequence of elements in the set $\{x_1, \ldots, x_n\}$, for some $n \geq 0$.
3. Each constraint in a context P of an instance declaration $P \Rightarrow C\,\overline{\tau}$ must have the form $C\,a$, where a is a type variable occurring in $\overline{\tau}$.

Restriction 1 allows only single parameter type classes, but multi-parameter type classes are widely used by programmers and in Haskell libraries and are supported in many Haskell implementations. For example, consider type class *Map* parameterized by the key and element types, and the type class *Collection*, parameterized by the type constructor and the type of elements of the collection, partly sketched below:

```
class Eq a ⇒ Collection c a where
  empty:: c a
  insert, delete:: a -> c a -> c a
  member:: a -> c a -> Bool
  ...
```

`instance` *Show* `(`*Tree Int*`) where ...` is an example of an instance declaration that does not follow restriction (2), because the head of the constraint (which has an empty context) consists of type constructor *Tree* applied to *Int*, not to a type variable.

Flag `-XFlexibleInstances` can be used by GHC users to avoid enforcing condition (2), i.e. to allow the head of a constraint in an instance declaration to be arbitrarily nested. The next is an example that does not follow restriction (3), since $s\ a$ is not just a type variable: `instance` *Show* `(`*s a*`) ⇒` *Show* `(`*Sized s a*`)`...

Instances that do not follow these restrictions are common in Haskell programs, specially in the presence of multi-parameter type classes.

Flag `-XFlexibleContexts` can be used by GHC users to avoid restriction (3). With the use of this flag, contexts are restricted as follows:

1. No type variable can have more occurrences in a constraint of a context than in the head.
2. The sum of the number of *occurrences* of type variables and type constructors in a context must be smaller than in the head.

This restriction is known as the *Paterson Condition*. In some cases, it is still over restrictive. As an example, consider the following code:

```
data Rose f a = Rose (f (Rose f a))

instance (Show (f (Rose f a)), Show a) ⇒
                  Show (Rose f a) where ...
```

This instance of *Show* is rejected by GHC because it has more occurrences of type variable f in a constraint than in the head. Flag `-XUndecidableInstances`, which lifts all restrictions (including those related to the use of functional dependencies), is needed to compile this code. With this flag, termination is ensured by imposing a depth limit on a recursion stack [20].

3 Constrained polymorphism and type class constraints

The Haskell type class system is based on the more general theory of *qualified types* [12], which extends the Hindley-Milner type system with constrained types.

The syntax of types with type class constraints is defined in Figure 1, where meta-variable usage is also indicated. For simplicity, and following common practice, kinds are not considered explicitly in type expressions, and type applications are assumed to be well kinded. Function types $\tau_1 \to \tau_2$ are constructed as the curried application of the function type constructor to two arguments, and are written as usual in infix notation.

The union of constraint sets P and Q is denoted by P, Q and a slight abuse of notation is made by writing simply π for the singleton constraint set $\{\pi\}$.

Function tv is overloaded, yielding the set of free type variables of types, constraints or constraint sets, and is defined as usual. Sequence $\overline{\alpha}$ used in the context of a set denotes of course the set of type variables in the sequence. The set of constraint axioms

Type constructor	T	
Class name	C	
Type variable	α, β	
Type	τ	$::= \alpha \mid T \mid \tau\ \tau$
Type constraint	π	$::= C\,\overline{\tau}$
Constraint set	Q, P	$::= \emptyset \mid \pi, Q$
Type scheme	σ	$::= \forall\,\overline{\alpha}.\, Q \Rightarrow \tau$
Constraint axioms Θ		$::= \emptyset \mid \forall\,\overline{\alpha}.\, Q \Rightarrow \pi,\ \Theta$

Fig. 1 Constrained types and Context

$$\boxed{\Theta, P \Vdash Q}$$

$$\frac{Q \subseteq P}{\Theta, P \Vdash Q}\ \texttt{Mono} \qquad \frac{\Theta, P \Vdash P' \quad \Theta, Q \Vdash Q'}{\Theta, P, Q \Vdash P', Q'}\ \texttt{Conj}$$

$$\frac{\Theta, P \Vdash Q}{\Theta, SP \Vdash SQ}\ \texttt{Subst} \qquad \frac{(\forall\,\overline{\alpha}.\, P \Rightarrow \pi) \in \Theta}{\Theta, P \Vdash \pi}\ \texttt{Inst}$$

$$\frac{\Theta, P \Vdash Q' \quad \Theta, Q' \Vdash Q}{\Theta, P \Vdash Q}\ \texttt{Trans}$$

Fig. 2 Type Class Constraint Entailment

Θ is induced by class and instance declarations of a program. Each instance declaration `instance` $P \Rightarrow \pi$ `where ...` introduces an axiom scheme $\forall\,\overline{\alpha}.\, P \Rightarrow \pi$, where $\overline{\alpha} = tv(P \Rightarrow \pi)$.

For simplicity and to avoid clutter, in this paper constraint axioms introduced by type class declarations are not considered, since they add no additional problems with respect to termination of constraint set satisfiability and simplification algorithms.

The entailment relation for type class constraints is defined in Figure 2. Rule (`Mono`) expresses the property of monotonicity, (`Conj`) of transitivity, (`Subst`) of closure under type substitution (cf. [12]), (`Inst`) defines entailment according to a constraint axiom and (`Conj`) deals with sets with more than one constraint.

A type substitution S is a (kind-preserving) function from type variables to types, and extends straightforwardly to constraints, and to sets of types and sets of constraints. For convenience, a substitution is often written as a finite mapping $[\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n]$, which is also abbreviated as $[\overline{\alpha} \mapsto \overline{\tau}]$. Juxtaposition $S'S$ is used as a synonym for function composition, $S' \circ S$, the domain of a substitution S is defined by $dom(S) = \{\alpha \mid S(\alpha) \neq \alpha\}$ and the restriction of S to V is given by $S|_V(\alpha) = S(\alpha)$ if $\alpha \in V$, otherwise α.

3.1 Constraint Set Satisfiability

Constraint set satisfiability is central to the interpretation of constrained types and is closely related to simplification and improvement. Following [13], $\lfloor P \rfloor_\Theta$ denotes the set of satisfiable instances of constraint set P, with respect to constraint axioms Θ:

$$\lfloor P \rfloor_\Theta = \{\, SP \mid \Theta \Vdash SP \,\}$$

Equality of constraint sets is considered modulo type variable renaming. That is, constraint sets P and Q are considered to be equal by considering also that a renaming substitution S can be applied to P so as to make $S\,P$ and Q equal. A substitution S is a renaming substitution if for all $\alpha \in dom(S)$ we have that $S(\alpha) = \beta$, for some type variable $\beta \notin dom(S)$.

If $SP \in \lfloor P \rfloor_\Theta$ then S is called a satisfying substitution for P.

Subscript Θ will not be used hereafter because satisfiability is always considered with respect to a set of global constraint axioms Θ.

For any substitution S and constraint set P we have that $\lfloor SP \rfloor \subseteq \lfloor P \rfloor$. The reverse inclusion, $\lfloor P \rfloor \subseteq \lfloor SP \rfloor$, does not always hold, and allow us to characterize improvement of the set of constraints P to an equivalent but simpler or more informative constraint set SP, such that $\lfloor SP \rfloor = \lfloor P \rfloor$. Substitution S is called an improving substitution for P if applying S to P preserves the set of satisfiable instances, that is, if $\lfloor SP \rfloor = \lfloor P \rfloor$.

The next section presents constraint set satisfiability algorithms, including an algorithm that uses a criterion for guaranteeing termination on any given input. This termination criterion is used in section 5, to define a constraint set simplification algorithm.

4 Computing Constraint Set Satisfiability

Figure 3 presents a computable function that, given any constraint set P, returns, if it terminates, the set of satisfying substitutions for P. The definition uses judgements of the form $\Theta \vdash^{\texttt{sats}} P \rightsquigarrow \mathbb{S}$, meaning that $\mathbb{S}$ is the set of satisfying substitutions for P, with respect to constraint axioms Θ. The following function is used:

$$\begin{aligned} sats(\pi, \Theta) = \{\, & (S|_{tv(\pi)}, SP, \pi_0) \mid (\forall\,\overline{\alpha}.\, P_0 \Rightarrow \pi_0) \in \Theta, \\ & S_1 = [\overline{\alpha} \mapsto \overline{\beta}],\ \overline{\beta} \text{ fresh}, \\ & (P \Rightarrow \pi') = S_1\,(P_0 \Rightarrow \pi_0), \\ & S = mgu(\pi = \pi') \} \end{aligned}$$

where function mgu gives a most general unifier for a pair of constraints, written as an equality. That is, $mgu(C\,\overline{\tau} = C\,\overline{\tau}')$ gives a substitution S such that, $S\,\overline{\tau} =$

$$\boxed{\Theta \vdash^{\mathtt{sats}} P \rightsquigarrow \mathbb{S}}$$

$$\frac{}{\Theta \vdash^{\mathtt{sats}} \emptyset \rightsquigarrow \{id\}}\ \mathtt{SEmpty}$$

$$\frac{\Theta \vdash^{\mathtt{sats}} \pi \rightsquigarrow \mathbb{S}_0 \quad \mathbb{S} = \{S'S \mid S \in \mathbb{S}_0,\, S' \in \mathbb{S}_1,\, \Theta \vdash^{\mathtt{sats}} SP \rightsquigarrow \mathbb{S}_1\}}{\Theta \vdash^{\mathtt{sats}} \pi, P \rightsquigarrow \mathbb{S}}\ \mathtt{SConj}$$

$$\frac{\Delta = sats(\pi, \Theta) \quad \mathbb{S} = \{S'S \mid (S, Q, \pi') \in \Delta,\, S' \in \mathbb{S}_0,\, \Theta \vdash^{\mathtt{sats}} Q \rightsquigarrow \mathbb{S}_0\}}{\Theta \vdash^{\mathtt{sats}} \pi \rightsquigarrow \mathbb{S}}\ \mathtt{SInst}$$

Fig. 3 Constraint Set Satisfiability

$S\,\overline{\tau}'$ and, for any S' such that $S'\,\overline{\tau} = S'\,\overline{\tau}'$, it holds that $S' = S'' \circ S$, for some S''.[1]

Let $\mathbb{S}$ be the returned set of satisfying substitutions for a given constraint P. Since $S \in \mathbb{S}$ implies $dom(S) \subseteq tv(P)$ — because if S is in $sats(\pi, \Theta)$ then $dom(S) \subseteq tv(\pi)$ —, the only possible satisfying substitution to be returned for the empty set of constraints is the identity substitution (id), as defined by rule `SEmpty`. Rule `SInst` computes the set $\mathbb{S}_0$ of satisfying substitutions $S \in \mathbb{S}_0$ for a given constraint π, by determining the set of constraint axioms $\forall\,\overline{\alpha}.\, P_0 \Rightarrow \pi_0$ in Θ such that π unifies with π_0, and composing these substitutions with those obtained by recursively computing the set of satisfying substitutions for contexts $S\,P_0$. Rule `SConj` deals with sets of constraints. The following examples illustrate the use of these rules.

`B`, `I` and `F` are used in the sequel as abbreviations of *Bool*, *Int* and *Float*, respectively.

Example 1 Consider $P = \{A\,a\,b,\, D\,b\}$ and

$$\Theta = \{A\,\mathtt{I}\,\mathtt{[I]}, A\,\mathtt{I}\,\mathtt{[B]}, C\,\mathtt{I}, \forall b.\, C\,b \Rightarrow D\,\mathtt{[}b\mathtt{]}\}$$

Satisfiability of P with respect to Θ yields a set of substitutions $\mathbb{S}$ given by:

$$\frac{\Theta \vdash^{\mathtt{sats}} A\,a\,b \rightsquigarrow \mathbb{S}_0 \quad \mathbb{S} = \{S'S \mid S \in \mathbb{S}_0,\, S' \in \mathbb{S}_1, \Theta \vdash^{\mathtt{sats}} S(D\,b) \rightsquigarrow \mathbb{S}_1\}}{\Theta \vdash^{\mathtt{sats}} A\,a\,b, \{D\,b\} \rightsquigarrow \mathbb{S}}\ \mathtt{SConj}$$

Then:

$$\frac{\Delta_0 = \{(S_1, \emptyset, A\,\mathtt{I}\,\mathtt{[I]}), (S_2, \emptyset, A\,\mathtt{I}\,\mathtt{[B]})\} \quad \mathbb{S}_0 = \{S'S \mid (S, Q, \pi') \in \Delta_0,\, S' \in \mathbb{S}', \Theta \vdash^{\mathtt{sats}} Q \rightsquigarrow \mathbb{S}'\}}{\Theta \vdash^{\mathtt{sats}} A\,a\,b \rightsquigarrow \mathbb{S}_0}\ \mathtt{SInst}$$

[1] See, for example, [2], for the general theory of unification and algorithms for computing a most general unifier for a set of term equalities.

where $S_1 = [a \mapsto \mathtt{I},\, b \mapsto \mathtt{[I]}]$, $S_2 = [a \mapsto \mathtt{I},\, b \mapsto \mathtt{[B]}]$.

Then, by rule `SConj`, the set of satisfying substitutions for $S_1(D\,b) = D\,\mathtt{[I]}$ and $S_2(D\,b) = D\,\mathtt{[B]}$ must be computed, and are given respectively by:

$$\frac{\Delta_1 = \{(S_1'|_\emptyset, \{C\,\mathtt{I}\}, D\,\mathtt{[}b\mathtt{]})\} \quad \mathbb{S}_1^1 = \{S'S \mid (S, Q, \pi') \in \Delta_1,\, S' \in \mathbb{S}', \Theta \vdash^{\mathtt{sats}} Q \rightsquigarrow \mathbb{S}'\}}{\Theta \vdash^{\mathtt{sats}} D\,\mathtt{[I]} \rightsquigarrow \mathbb{S}_1^1}\ \mathtt{SInst}$$

where $S_1' = [b_1 \mapsto \mathtt{I}]$, b_1 is a fresh type variable, $S_1'|_\emptyset = id$, and

$$\frac{\Delta_2 = \{(S_2'|_\emptyset, \{C\,\mathtt{B}\}, D\,\mathtt{[}b\mathtt{]})\} \quad \mathbb{S}_1^2 = \{S'S \mid (S, Q, \pi') \in \Delta_2, S' \in \mathbb{S}', \Theta \vdash^{\mathtt{sats}} Q \rightsquigarrow \mathbb{S}'\}}{\Theta \vdash^{\mathtt{sats}} D\,\mathtt{[B]} \rightsquigarrow \mathbb{S}_1^2}\ \mathtt{SInst}$$

where $S_2' = [b_2 \mapsto \mathtt{B}]$, b_2 is a fresh type variable, $S_2'|_\emptyset = id$. Now, $\mathbb{S}_1^1 = \{id\}$ and $\mathbb{S}_1^2 = \emptyset$. Thus, $\mathbb{S} = \{S_1\}$.

The example below, extracted from [3], illustrates non-termination of the computation of the set of satisfying substitutions by the function defined in Figure 3. We use $T^2\,\tau$ to abbreviate $T(T\,\tau)$ and similarly for other indices greater than 2.

Example 2 Let $\Theta = \{\forall a, b.\, \{C\,a\,b\} \Rightarrow C\,(T^2\,a)\,b\}$ and consider computing satisfiability of $\pi = C\,a\,(T\,a)$ with respect to Θ.

We have that π unifies with the head of constraint axiom $\forall a, b.\, (C\,a\,b) \Rightarrow C\,(T^2\,a)\,b$, giving substitution $S = [a \mapsto T^2\,a_1,\, b_1 \mapsto T^3\,a_1]$. We must then recursively compute the set of satisfying substitutions of constraint $S(C\,a_1\,b_1) = C\,a_1\,(T^3\,a_1)$. This constraint also unifies with $\forall a, b.\, (C\,a\,b) \Rightarrow C\,(T^2\,a)\,b$, giving substitution $S_1 = [a_1 \mapsto (T^2\,a_2), b_2 \mapsto (T^3 a_1 = T^5\,a_2)]$. Again, we must recursively compute the set of satisfying substitutions of constraint $S_1(C\,a_2\,b_2) = C\,a_2\,(T^5\,a_2)$, and the process goes on forever.

The following theorems state respectively correctness and completeness of the constraint set satisfiability algorithms presented in Figure 3, with respect to the entailment relation.

Theorem 1 (Correctness of $\vdash^{\mathtt{sats}}$) *If $\Theta \vdash^{\mathtt{sats}} P \rightsquigarrow \mathbb{S}$ then $\Theta \Vdash S\,P$, for all $S \in \mathbb{S}$.*

Proof: By induction over the derivation of $\Theta \vdash^{\mathtt{sats}} P \rightsquigarrow \mathbb{S}$. The only interesting case is for rule `SInst`. Let $\pi = C\,\overline{\tau}$ and $\Delta = sats(\pi, \Theta)$. If $\Delta = \emptyset$, the theorem holds trivially. Thus, assume $\Delta \neq \emptyset$ and let $(S, Q, C\,\overline{\tau}_0) \in \Delta$. By the definition of *sats*, this means that $\forall\,\overline{\alpha}.\, P_0 \Rightarrow C\,\overline{\tau}_0 \in \Theta$, where $\overline{\alpha} = tv(P_0 \Rightarrow C\,\overline{\tau}_0)$, and $P' \Rightarrow$

$C\,\overline{\tau} = [\,\overline{\alpha} \mapsto \overline{\beta}\,]P_0 \Rightarrow C\,\overline{\tau}_0$. By rule `Inst` we have that $\Theta,\ P_0 \Vdash C\,\overline{\tau}_0$ is provable. We also have that $\Theta \vdash^{\mathtt{sats}} Q \rightsquigarrow \mathbb{S}_0$, where $Q = S\,[\,\overline{\alpha} \mapsto \overline{\beta}\,]\,P_0$, and thus, by the induction hypothesis, we have that (1) $\Theta \Vdash S'\,Q$ holds for all $S' \in \mathbb{S}_0$. Also, since $\Theta,\ P_0 \Vdash C\,\overline{\tau}_0$ is provable, we have, by rule `Subst`, that (2) $\Theta,\ S_0\,P_0 \Vdash S_0\,C\,\overline{\tau}_0$, where $S_0 = S'\,S\,[\,\overline{\alpha} \mapsto \overline{\beta}\,]$. From (1) and (2) we have, by rule `Trans`, that $\Theta \Vdash S_0\,C\,\overline{\tau}_0$ is provable. Since $S\,\overline{\tau} = S\,[\,\overline{\alpha} \mapsto \overline{\beta}\,]\,\overline{\tau}_0$, this means that $\Theta \Vdash S'\,S\,C\,\overline{\tau}$ is provable. □

Theorem 2 (Completeness of $\vdash^{\mathtt{sats}}$) *If $\Theta \Vdash S\,P$ then there exist $S' \in \mathbb{S}$ and S'' such that $S''\,S'\,P = S\,P$, where $\Theta \vdash^{\mathtt{sats}} P \rightsquigarrow \mathbb{S}$.*

Proof: Induction over $S\,P$ in $\Theta \Vdash S\,P$. □

4.1 Termination

The algorithm presented in Figure 3 is modified in this section in order to ensure termination on any given input. The basic idea is to associate a value to each constraint head of the set of constraint axioms that is unified with some constraint in the recursive process of computing satisfiability, and require that the value associated to a constraint head always decreases in a new unification that occurs during this process. Computation stops if this requirement is not fulfilled, with no satisfying substitution found for the original set of constraints. Values in this decreasing chain are a measure of the size of types in constraints that unify with each constraint head axiom: the size of each constraint in this chain is decreasing or there exists a position of a type argument in the constraint such that the type's size is decreasing.

Let the constraint value $\eta(\pi)$ of a constraint π, which gives the number of occurrences of type variables and type constructors in π, be defined as follows:

$$\begin{array}{ll}\eta(C\,\tau_1 \cdots \tau_n) & = \sum_{i=1}^{n} \eta(\tau_i)\\ \eta(T) & = 1\\ \eta(\alpha) & = 1\\ \eta(\tau_1\,\tau_2) & = \eta(\tau_1) + \eta(\tau_2)\end{array}$$

A finite constraint-head-value function Φ is used to map constraint heads π_0 of Θ to pairs (I, Π), as follows.

The first component I is a tuple $(v_0, ..., v_n)$, where v_0 is the least $\eta(S\pi')$ of all constraints π' that have unified with π_0 during the satisfiability test for π, where $S = mgu(\pi'_0, \pi')$. Each v_i, $1 \leq i \leq n$, is the least $\eta(\tau_i)$ where τ_i is a type belonging to some $S\pi'$ that has unified with π_0.

We let $I.v_i$ denote the i-th value of I and, similarly, $\Phi(\pi_0).I$ and $\Phi(\pi_0).\Pi$ denote respectively the first and second components of $\Phi(\pi_0)$.

The second component Π of $\Phi(\pi_0)$ contains constraints π' that unify with π_0 and have constraint values equal to v_0. This allows distinct constraints with equal constraint values to unify with π_0 (cf. Example 6 below).

Consider a recursive step in a test of satisfiability where a constraint π unifies with a constraint head $\pi_0 = C\,\tau_1 \ldots \tau_n$, with $S = mgu(\pi_0, \pi)$. Let $\Phi(\pi_0) = ((v_0, ..., v_n), \Pi)$ and $\eta(S\pi) = n_0$. $\Phi(\pi_0)$ is then updated as follows. If $n_0 < v_0$ then only the value v_0 is updated, to n_0. In the case that $n_0 = v_0$ and $\pi \notin \Pi$, $\Phi(\pi_0)$ is updated to $((v_0, ..., v_n), \Pi \cup \{S\pi\})$, i.e. we include $S\pi$ in the set of constraints that have the same value v_0. Finally, if $n_0 > v_0$, we set v_0 to -1 and for each τ_i such that $\eta(\tau_i) \geq v_i$, we update v_i with -1, otherwise v_i is updated with $\eta(\tau_i)$. In subsequent steps for constraints π' that unify with π_0, with S' as a unifying substitution, it is required that $\eta(S'\tau_i) < v_i$; if there's no such i, a failure in the termination criteria is detected.

Let $f[x \mapsto y]$ denote the usual function updating notation for f' given by $f'(x') = y$ if $x' = x$, otherwise $f(x)$.

We define $\Phi[\pi_0, \pi]$ as updating of $\Phi(\pi_0) = (I, \Pi)$ as follows, where $I = (v_0, v_1, \ldots, v_n)$, $\pi = C\,\tau_1 \cdots \tau_n$, $n_0 = \eta(\pi)$:

$$\begin{array}{rl}\Phi[\pi_0, \pi] = & \Phi[\pi_0 \mapsto ((n_0, v_1, \ldots, v_n), \Pi)] \text{ if } n_0 < I.v_0;\\ & \Phi[\pi_0 \mapsto (I, \Pi \cup \{\pi\})] \text{ if } n_0 = I.v_0, \pi \notin \Pi;\\ & \Phi[\pi_0 \mapsto (I', \Pi)] \text{ if } n_0 > I.v_0, \exists\, i.\, (I'.v_i \neq -1)\\ & \quad \text{where, for } i = 1, \ldots, n,\\ & \qquad I'.v_i = \begin{cases} -1 & \text{if } I.v_i < \eta(\tau_i) \text{ or } i = 0\\ \eta(\tau_i) & \text{otherwise}\end{cases}\\ & \textit{Fail} \text{ otherwise}\end{array}$$

The computable function (`tsat`) for constraint satisfiability, defined in Figure 4, uses judgements of the form $\Theta, \Phi \vdash^{\mathtt{tsat}} P \rightsquigarrow \mathbb{S}$, with constraint-head-value function Φ as additional parameter.

The set of satisfying substitutions for constraint set P with respect to the set of constraint axioms Θ is given by $\mathbb{S}$, such that $\Theta, \Phi_0 \vdash^{\mathtt{tsat}} P \rightsquigarrow \mathbb{S}$ holds, where $\Phi_0(\pi_0) = (I_0, \emptyset)$ for each constraint head $\pi_0 = C\,\tau_1 \ldots \tau_n$ in Θ and I_0 is a tuple formed by $n+1$ occurrences of a large enough integer constant, represented by ∞.

Consider the following.

Example 3 Consider computing satisfiability of $\pi = Eq\,\mathtt{[[I]]}$ in $\Theta = \{Eq\ \mathtt{I},\ \forall a.\, Eq\ a \Rightarrow Eq\,\mathtt{[}a\mathtt{]}\}$, letting $\pi_0 = Eq\,\mathtt{[}a\mathtt{]}$; we have:

$$\boxed{\Theta, \Phi \vdash^{\mathtt{tsat}} P \rightsquigarrow \mathbb{S}}$$

$$\frac{}{\Theta, \mathit{Fail} \vdash^{\mathtt{tsat}} \emptyset \rightsquigarrow \emptyset}\ \mathtt{SFail}_1 \qquad \frac{}{\Theta, \Phi \vdash^{\mathtt{tsat}} \emptyset \rightsquigarrow \{id\}}\ \mathtt{SEmpty}_1$$

$$\frac{\Theta, \Phi \vdash^{\mathtt{tsat}} \pi \rightsquigarrow \mathbb{S}_0 \quad \mathbb{S} = \{S'S \mid S \in \mathbb{S}_0,\ S' \in \mathbb{S}_1,\ \Theta, \Phi \vdash^{\mathtt{tsat}} SP \rightsquigarrow \mathbb{S}_1\}}{\Theta, \Phi \vdash^{\mathtt{tsat}} \pi, P \rightsquigarrow \mathbb{S}}\ \mathtt{SConj}_1$$

$$\frac{\Delta = \mathit{sats}(\pi, \Theta) \quad \mathbb{S} = \{S'S \mid (S, Q, \pi') \in \Delta,\ S' \in \mathbb{S}_0,\ \Phi' = \Phi[\pi', S\pi],\ \Theta, \Phi' \vdash^{\mathtt{tsat}} Q \rightsquigarrow \mathbb{S}_0\}}{\Theta, \Phi \vdash^{\mathtt{tsat}} \pi \rightsquigarrow \mathbb{S}}\ \mathtt{SInst}_1$$

Fig. 4 Terminating Constraint Set Satisfiability

$$\frac{\begin{array}{l}\Delta_0 = \mathit{sats}(\pi, \Theta) = \{(S|_\emptyset, \{Eq\,\mathtt{[I]}\}, \pi_0)\} \\ S = [a_1 \mapsto \mathtt{[I]}] \\ \mathbb{S}_0 = \{S_1 \circ id \mid S_1 \in \mathbb{S}_1,\ \ \Theta, \Phi_1 \vdash^{\mathtt{tsat}} Eq\,\mathtt{[I]} \rightsquigarrow \mathbb{S}_1\}\end{array}}{\Theta, \Phi_0 \vdash^{\mathtt{tsat}} \pi \rightsquigarrow \mathbb{S}_0}$$

where $\Phi_1 = \Phi_0[\pi_0, \pi]$, $\Phi_1(\pi_0).I = (\eta(\pi) = 3, \infty)$, $S\pi = \pi$ and a_1 is a fresh type variable; then:

$$\frac{\begin{array}{l}\Delta_1 = \mathit{sats}(Eq\,\mathtt{[I]}, \Theta) = \{(S'|_\emptyset, \{Eq\,\mathtt{I}\}, \pi_0)\} \\ S' = [a_2 \mapsto \mathtt{I}] \\ \mathbb{S}_1 = \{S_2 \circ id \mid S_2 \in \mathbb{S}_2,\ \ \Theta, \Phi_2 \vdash^{\mathtt{tsat}} Eq\,\mathtt{I} \rightsquigarrow \mathbb{S}_2\}\end{array}}{\Theta, \Phi_1 \vdash^{\mathtt{tsat}} Eq\,\mathtt{[I]} \rightsquigarrow \mathbb{S}_1}$$

where $\Phi_2 = \Phi_1[\pi_0, Eq\,\mathtt{[I]}]$ and $\eta(Eq\,\mathtt{[I]}) = 2$ is less than $\Phi_1(\pi_0).I.v_0 = 3$; then:

$$\frac{\begin{array}{l}\Delta_2 = \mathit{sats}(Eq\,\mathtt{I}, \Theta) = \{(id, \emptyset, Eq\,\mathtt{I})\} \\ \mathbb{S}_2 = \{S_3 \circ id \mid S_3 \in \mathbb{S}_3,\ \ \Theta, \Phi_3 \vdash^{\mathtt{tsat}} \emptyset \rightsquigarrow \mathbb{S}_3 = \{id\}\}\end{array}}{\Theta, \Phi_2 \vdash^{\mathtt{tsat}} Eq\,\mathtt{I} \rightsquigarrow \mathbb{S}_2}$$

where $\Phi_3 = \Phi_2[Eq\,\mathtt{I}, Eq\,\mathtt{I}]$, $\mathbb{S}_3 = \{id\}$ by ($\mathtt{SEmpty}_1$).

Example 4 Consider again Example 2: we want to obtain the set of satisfying substitutions for constraint $\pi = C\ a\ (T\ a)$, given $\Theta = \{\forall a, b.\, C\ a\ b \Rightarrow C\ (T^2\ a)\ b\}$ (computation with input π by the function in Figure 3 does not terminate). We have, where $\pi_0 = C\ (T^2\ a)\ b$:

$$\frac{\begin{array}{l}\Delta_0 = \mathit{sats}(\pi, \Theta) = \{(S\,|_{\{a\}}, \{\pi_1\}, \pi_0)\} \\ S = [a \mapsto T^2\ a_1, b_1 \mapsto T^3\ a_1] \\ \pi_1 = C\ a_1\ (T^3\ a_1) \\ \mathbb{S}_0 = \{S_1 \circ [a \mapsto T^2\ a_1] \mid S_1 \in \mathbb{S}_1,\ \ \Theta, \Phi_1 \vdash^{\mathtt{tsat}} \pi_1 \rightsquigarrow \mathbb{S}_1\}\end{array}}{\Theta, \Phi_0 \vdash^{\mathtt{tsat}} \pi \rightsquigarrow \mathbb{S}_0}$$

where $\Phi_1 = \Phi_0[\pi_0, S\pi]$, $\eta(S\pi) = \eta(C\ (T^2\ a_1)\ (T^3\ a_1)) = 7 < \Phi_0(\pi_0).I.v_0 = \infty$; then:

$$\frac{\begin{array}{l}\Delta_1 = \mathit{sats}(\pi_1, \Theta) = \{(S'\,|_{\{a_1\}},\ \{\pi_2\}, \pi_0)\} \\ S' = [a_1 \mapsto T^2\ a_2, b_2 \mapsto T^3\ a_1 = T^5 a_2] \\ \pi_2 = C\ a_2\ (T^5\ a_2) \\ \mathbb{S}_1 = \{S_2 \circ [a_1 \mapsto T^2\ a_2] \mid S_2 \in \mathbb{S}_2,\ \ \Theta, \Phi_2 \vdash^{\mathtt{tsat}} \pi_2 \rightsquigarrow \mathbb{S}_2\}\end{array}}{\Theta, \Phi_1 \vdash^{\mathtt{tsat}} \pi_1 \rightsquigarrow \mathbb{S}_1}$$

where $\Phi_2 = \Phi_1[\pi_0, S'\pi_1]$, $S'\pi_1 = (C\ (T^2\ a_2)\ (T^5\ a_2)$ and, since $\eta(S'\pi_1) = 9 > \Phi_1(\pi_0).I.v_0 = 7$, we have that $\Phi_2(\pi_0).I = (-1, \eta(T^2\ a_2) = 3, \eta(T^5\ a_2) = 6)$; then:

$$\frac{\begin{array}{l}\Delta_1 = \mathit{sats}(\pi_2, \Theta) = \{(S'\,|_{\{a_2\}},\ \{\pi_3\}, \pi_0)\} \\ S'' = [a_2 \mapsto T^2\ a_3, b_3 \mapsto T^5\ a_2 = T^7 a_3] \\ \pi_3 = C\ a_3\ (T^7\ a_3) \\ \mathbb{S}_2 = \{S_3 \circ [a_2 \mapsto T^2\ a_3] \mid S_3 \in \mathbb{S}_3,\ \ \Theta, \Phi_3 \vdash^{\mathtt{tsat}} \pi_3 \rightsquigarrow \mathbb{S}_3\}\end{array}}{\Theta, \Phi_1 \vdash^{\mathtt{tsat}} \pi_1 \rightsquigarrow \mathbb{S}_2}$$

where $\Phi_3 = \Phi_2[\pi_0, S''\pi_2] = \mathit{Fail}$, because $\eta(S''\pi_2) = \eta(C\ (T^3\ a_3)\ (T^7\ a_3)) = 12 > \Phi_2(\pi_0).I.v_0 = 9$ and there's no i such that $\Phi_3(\pi_0).I.v_i \neq -1$, meaning that no parameter of $S''\pi_2$ has a decreasing η value.

The following illustrates an example of a satisfiable constraint for which computation of satisfiability involves computing satisfiability of constraints π' that unify with a constraint head π_0 such that $\eta(\pi')$ is greater than the upper bound associated to π_0.

Example 5 Consider satisfiability of $\pi = C\ \mathtt{I}\ (T^3\ \mathtt{I})$ in $\Theta = \{C\ (T\ a)\ \mathtt{I}, \forall a, b.\, C\ (T^2\ a)\ b \Rightarrow C\ a\ (T\ b)\}$. We have, where $\pi_0 = C\ a\ (T\ b)$:

$$\frac{\begin{array}{l}\Delta_0 = \mathit{sats}(\pi, \Theta) = \{(S\,|_\emptyset, \{\pi_1\}, \pi_0)\} \\ S = [a_1 \mapsto \mathtt{I}, b_1 \mapsto T^2\ \mathtt{I}] \\ \pi_1 = C\ (T^2\ \mathtt{I})\ (T^2\ \mathtt{I}) \\ \mathbb{S}_0 = \{S_1 \circ id \mid S_1 \in \mathbb{S}_1,\ \ \Theta, \Phi_1 \vdash^{\mathtt{tsat}} \pi_1 \rightsquigarrow \mathbb{S}_1\}\end{array}}{\Theta, \Phi_0 \vdash^{\mathtt{tsat}} \pi \rightsquigarrow \mathbb{S}_0}$$

where $\Phi_1 = \Phi_0[\pi_0, \pi]$, $\eta(\pi) = 5 < \Phi_0(\pi_0).I.v_0 = \infty$, $S\pi = \pi$; then:

$$\frac{\begin{array}{l}\Delta_1 = \mathit{sats}(\pi_1, \Theta) = \{(S'\,|_\emptyset, \{\pi_2\}, \pi_0)\} \\ S' = [a_2 \mapsto T^2\ \mathtt{I}, b_2 \mapsto T\ \mathtt{I}] \\ \pi_2 = C\ (T^4\ \mathtt{I})\ (T\ \mathtt{I}) \\ \mathbb{S}_1 = \{S_2 \circ [a_1 \mapsto T^2\ a_2] \mid S_2 \in \mathbb{S}_2,\ \ \Theta, \Phi_2 \vdash^{\mathtt{tsat}} \pi_2 \rightsquigarrow \mathbb{S}_2\}\end{array}}{\Theta, \Phi_1 \vdash^{\mathtt{tsat}} \pi_1 \rightsquigarrow \mathbb{S}_1}$$

where $\Phi_2 = \Phi_1[\pi_0, \pi_1]$, $\Phi_1(\pi_0).I = (5, \infty, \infty)$, $S'\pi_1 = \pi_1$. Since $\eta(\pi_1) = 6 > 5 = \Phi_1(\pi_0).I.v_0$, we have that $\Phi_2(\pi_0).I$ becomes equal to $(-1, 3, 3)$.

Then, consider that $\pi_2 = C\ \tau_1\ \tau_2$ where $\tau_1 = T^4\ I$ and $\tau_2 = T\ I$. Since $\eta(\pi_2) > \Phi_2(\pi_0).I.v_0 = -1$, there must exist i, $1 \leq i \leq 2$, such that $\eta(\tau_i) < \Phi_2(\pi_0).v_i$, and

such condition is satisfied for $i = 2$, updating $\Phi_2(\pi_0).I$ to $(-1, -1, 2)$. Satisfiability is then finally tested for $\pi_3 = C\,(T^6\,\mathtt{I})\mathtt{I}$, that unifies with $\pi_0 = C\,(T\,a)\,\mathtt{I}$, which returns $\mathbb{S}_3 = \{[a_3 \mapsto T^5\,\mathtt{I}]|_\emptyset\} = \{id\}$. Constraint π is thus satisfiable, with $\mathbb{S}_0 = \{id\}$.

The following example illustrates the use of a set of constraints as a component of the constraint-head-value function.

Example 6 Let $\pi = C\,(T^2\,\mathtt{I})\,\mathtt{F}$, $\pi_0 = C\,(T\,a)\,b$, $\Theta = \{C\,\mathtt{I}\,(T^2\,\mathtt{F}), \forall a, b.\, C\,a\,(T\,b) \Rightarrow C\,(T\,a)\,b\}$:

$$\frac{\begin{array}{l}\Delta_0 = sats(\pi, \Theta) = \{(S\,|_\emptyset, \{\pi_1\}, \pi_0)\} \\ S = [a_1 \mapsto (T\,\mathtt{I}), b_1 \mapsto \mathtt{F}],\ \ \pi_1 = C\,(T\,\mathtt{I})\,(T\,\mathtt{F}) \\ \mathbb{S}_0 = \{S_1 \circ id \mid\ S_1 \in \mathbb{S}_1,\ \ \Theta, \Phi_1 \vdash^{tsat} \pi_1 \rightsquigarrow \mathbb{S}_1\}\end{array}}{\Theta, \Phi_0 \vdash^{tsat} \pi \rightsquigarrow \mathbb{S}_0}$$

where $\Phi_1 = \Phi_0[\pi_0, \pi]$, $S\pi = \pi$; then:

$$\frac{\begin{array}{l}\Delta_1 = sats(\pi_1, \Theta) = \{(S'\,|_\emptyset, \{\pi_2\}, \pi_0)\} \\ S' = [a_2 \mapsto \mathtt{I}, b_2 \mapsto T\,\mathtt{F}],\ \ \pi_2 = C\,\mathtt{I}\,(T^2\,\mathtt{F}) \\ \mathbb{S}_1 = \{S_2 \circ id \mid\ S_2 \in \mathbb{S}_2,\ \ \Theta, \Phi_2 \vdash^{\mathtt{tsat}} \pi_2 \rightsquigarrow \mathbb{S}_2\}\end{array}}{\Theta, \Phi_1 \vdash^{\mathtt{tsat}} \pi_1 \rightsquigarrow \mathbb{S}_1}$$

where $\Phi_2 = \Phi_1[\pi_0, \pi_1]$, $\eta(\pi_1) = 4 = \Phi_1(\pi_0).I.v_0 = \eta(\pi)$, $S'\pi_1 = \pi_1$ and π_1 is not in $\Phi_1(\pi_0).\Pi_1 = \emptyset$. We have that $\mathbb{S}_2 = \{id\}$, because $sats(C\,\mathtt{I}\,(T^2\,\mathtt{F}), \Theta) = \{(id, \emptyset, C\,\mathtt{I}\,(T^2\,\mathtt{F}))\}$, and π is then satisfiable.

Since satisfiability of type class constraints is in general undecidable [6], there exist instances of this problem for which our algorithm incorrectly reports unsatisfiability. An example that exhibits an incorrect behavior, constructed by encoding a solvable Post Correspondence Problem (PCP) instance by means of constraint set satisfiability, using G. Smith's scheme [6], is shown below. For all examples mentioned in the literature [15,17] and numerous tests that include those used by GHC involving pertinent GHC extensions, the algorithm works as expected, without the need of any compilation flag.

Example 7 This example uses a PCP instance taken from [9]. A PCP instance can be defined as composed of pairs of strings, each pair having a top and a bottom string, where the goal is to select a sequence of pairs such that the two strings obtained by concatenating top and bottom strings in such pairs are identical. The example uses three pairs of strings: $p_1 = (\mathtt{100}, \mathtt{1})$ (that is, pair 1 has string `100` as the top string and `1` as the bottom string), $p_2 = (\mathtt{0}, \mathtt{100})$ and $p_3 = (\mathtt{1}, \mathtt{00})$.

This instance has a solution: using numbers to represent corresponding pairs (i.e. `1` represents pair 1 and analogously for `2` and `3`), the sequence of pairs `1311322` is a solution.

A satisfiability problem that has a solution if and only if the PCP instance has a solution can be constructed by adapting G. Smith's scheme to Haskell's notation. We consider for this a two-parameter class C, and a constraint context such that $\Theta = \Theta_1 \cup \Theta_2 \cup \Theta_3$, where Θ_i is constructed from pair i, for $i = 1, 2, 3$:

$$\begin{array}{l}\Theta_1 = \{\, C\,(1 \to 0 \to 0)\,1, \\ \qquad \forall a, b.\, C\,a\,b \Rightarrow C\,(1 \to 0 \to 0 \to a)\,(1 \to b)\,\} \\ \Theta_2 = \{\, C\,0\,(1 \to 0 \to 0), \\ \qquad \forall a, b.\, C\,a\,b \Rightarrow C\,(0 \to a)\,(1 \to 0 \to 0 \to b)\,\} \\ \Theta_3 = \{\, C\,1\,(0 \to 0), \\ \qquad \forall a, b.\, C\,a\,b \Rightarrow C\,(1 \to a)\,(0 \to 0 \to b)\,\}\end{array}$$

We have that constraint $C\,a\,a$ is satisfiable, with a solution constructed from solution `1311322` of the PCP instance. Computation by our algorithm terminates, erroneously reporting unsatisfiability. The steps of the computation are omitted. The error occurs because a constraint $\pi_2 = C\,a_2\,(1 \to a_2)$ unifies with $\pi_{01} = C\,(1 \to 0 \to 0 \to a)\,(1 \to b)$ and $\eta(S\pi_2)$ is greater than $\Phi(\pi_{01}).I.v_0$, where $S = mgu(\pi_2, \pi_{01})$, and there's no $i \in \{1, 2\}$ such that $\Phi_3(\pi_0).I.v_i \neq -1$, meaning that no parameter of $S\pi_2$ has a decreasing η value.

To prove that the computation of the set of satisfying substitutions for any given constraint set P by the function defined in Figure 4 always terminates, consider that an infinite recursion might only occur if an infinite number of constraints unified with the head π_0 of one constraint axiom in Θ, since there exist finitely many constraint axioms in Θ. This is avoided because, for any new constraint π that unifies with π_0, we have, by the definition of $\Phi[\pi_0, \pi]$, that $\Phi(\pi_0)$ is updated to a value distinct from the previous ones (otherwise $\Phi[\pi_0, \pi]$ yields *Fail* and computation is stopped). The conclusion follows from the fact that $\Phi(\pi_0)$ can have only finitely many distinct values, for any π_0. This can be seen by considering that, for any π_0 such that $\Phi(\pi_0) = (I, \Pi)$, the insertion of a new constraint in Π decreases $k - k'$, where k is the finite number of all possible values that can be inserted in Π and k' is the cardinality of Π. Such a decrease causes then a decrease of Φ (since there exists only finitely many constraint heads π_0 in Θ). Similarly, at each step there must exist some i such that $I.v_i$ decreases, and this can happen only a finitely number of times. We conclude that computation on any given input terminates.

The proposed termination criteria is related to the *Paterson Condition* used in the GHC compiler (see Section 2). The constraint value is based on item 2 of this condition, but, instead of using it as a syntactic restriction over constraint heads and contexts in instance declarations, we use it in the definition of a finitely decreasing chain over recursively dependent constraints.

In comparison to the use of a recursion depth limit, our approach has the advantage that type-correctness is not implementation dependent (a constraint is or is not satisfiable with respect to a given set of constraint axioms). The use of a recursion depth limit can make a constraint set satisfiable in one implementation and unsatisfiable in another that uses a lower limit. Incorrectly reporting unsatisfiability can occur in both cases, but is expected to be extremely rare with our approach. We are not aware of any practical example where this occurs.

The main disadvantages of our approach are that it is not syntactically possible to characterize such incorrect unsatisfiability cases and it is not very easy for programmers to understand how type class constraints are handled in such a case, if and when it occurs. However, we expect these cases not to occur in practice.

The presented algorithm has been verified to behave correctly, without the need of any compilation flag, on all examples found in the literature [15], all GHC test cases, involving flags `FlexibleInstances`, `FlexibleContexts` and `UndecidableInstances`, and on Haskell libraries that use multi-parameter type classes, including the monad transformer library [1].

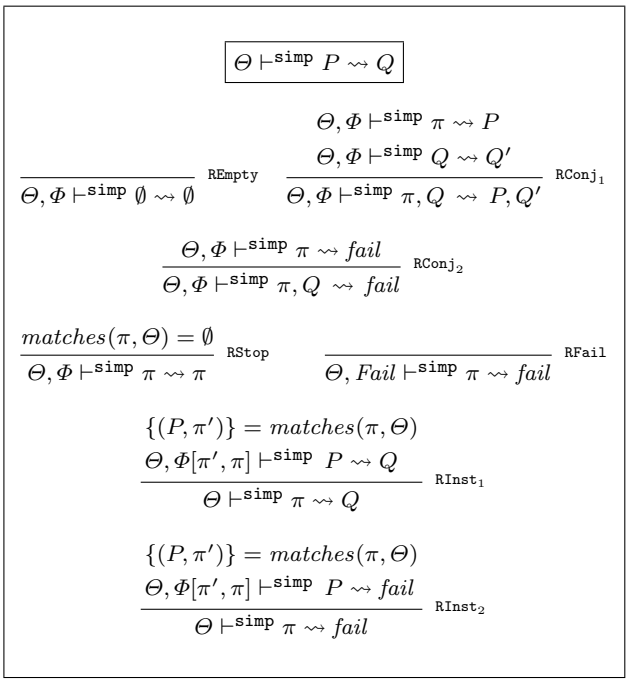

Fig. 5 Constraint Set Simplification

5 Constraint Set Simplification

The process of simplification of a constraint set, also called context reduction, consists of reducing each constraint π in this set to the context obtained by recursively reducing the context P of the *matching instance* for π in Θ, if such matching exists, until $P = \emptyset$ or there exists no instance in Θ that matches with π. In the latter case π reduces to itself.

This recursive process may not terminate: as a simple example, consider reduction of constraint $C\,a$ when $\Theta = \{\forall a.\, C\,a \Rightarrow C\,a\}$.

This section presents a computable function for constraint set simplification, where computation is guaranteed to terminate by using the same criterion used in Section 4.1.

Constraint set simplification is essentially based on instance matching. We use function $\mathit{matches}(\pi, \Theta)$, defined below, in order to capture the relevant information of matching constraint axioms in Θ with a given constraint π. Function *matches* is defined by using function *sats* (Section 4), through skolemization of type variables that occur in the given constraint argument (Skolem variables are non unifiable variables, that is, constants):

$$\mathit{matches}(\pi, \Theta) = \big\{(S\,P, \pi') \mid \Delta = \mathit{sats}\big([\overline{\alpha} \mapsto \overline{K}]\pi, \Theta\big),\ (S, S\,P, \pi') \in \Delta,\ \overline{\alpha} = \mathit{tv}(\pi),\ \overline{K} \text{ are fresh Skolem variables}\big\}$$

Function $\mathit{matches}(\pi, \Theta)$ returns either a singleton or an empty set[2].

Constraint set simplification uses a function defined in Figure 5 by means of judgements of the form $\Theta, \Phi \vdash^{\mathtt{simp}} P \rightsquigarrow Q$. This means that reduction of constraint set P under constraint axioms Θ either give constraint set Q as a result or fails. Failure is caused by the criterion used for ensuring termination, explained in Section 4.1. Using this function, context reduction is defined as follows, where Φ_0 is as defined in Section 4.1:

$$\frac{\text{for } i = 1, \ldots, n,\ Q_i = \begin{cases} \pi_i & \text{if } \Theta, \Phi_0 \vdash^{\mathtt{simp}} \pi_i \rightsquigarrow \mathit{fail} \\ Q_i' & \text{if } \Theta, \Phi_0 \vdash^{\mathtt{simp}} \pi_i \rightsquigarrow Q_i' \end{cases}}{\Theta \vdash^{\mathtt{simp}_0} \{\pi_1, \ldots \pi_n\} \rightsquigarrow Q_1, \ldots, Q_n}\ \mathtt{R}_0$$

The rules of Figure 5 are analogous to the ones in Figure 4, but now termination enforced by the termination criterion is reported as a failure, which must be propagated backwards along the recursive calls of

[2] We do not consider *overlapping instances* [20], since the subject is unrelated to termination of constraint set satisfiability and simplification. Supporting overlapping instances would need a modification of function *matches* so as to select a single instance if there exist overlapping matching instances.

the computation. Thus, reduction of a constraint π is now defined by two rules, ($\texttt{RInst}_1$) and ($\texttt{RInst}_2$) and, analogously, two different rules are used for specifying reduction of a non-singleton set of constraints.

Rule (`REmpty`) specifies that an empty set of constraints reduces to itself. Rule (`RStop`) specifies that a constraint π cannot be reduced if there is no instance in Θ that matches with π. Rule (`RFail`) enforces termination, expressing that reduction cannot be performed since updating of Φ fails.

The process of constraint set simplification is illustrated by the following example.

Example 8 Let $\Theta = \{\forall a.\, C\,(T\,a) \Rightarrow C\,a,\ D\,\mathtt{I}\}$ and $P = \{D\,\mathtt{I},\ C\,a\}$. According to rule ($\mathtt{R}_0$), reduction of P amounts to independently reducing constraints $D\,\mathtt{I}$ and $C\,a$.

Reduction of $D\,\mathtt{I}$ is defined by rule ($\texttt{RInst}_1$):

$$\frac{\{(\emptyset, D\,\mathtt{I})\} = \mathit{matches}(D\,\mathtt{I}, \Theta) \quad \Theta, \Phi_0[D\,\mathtt{I}, D\mathtt{I}] \vdash^{\mathit{simp}} \emptyset \rightsquigarrow \emptyset}{\Theta, \Phi_0 \vdash^{\mathit{simp}} D\,\mathtt{I} \rightsquigarrow \emptyset}$$

Reduction of $\pi = \pi_0 = C\,a$ results in failure, as shown below:

$$\frac{\{(C\,(T\,a_1), \pi_0)\} = \mathit{matches}(\pi, \Theta) \quad \Theta, \Phi_1 \vdash^{\mathit{simp}} (C\,(T\,a_1)) \rightsquigarrow \mathit{fail}}{\Theta, \Phi_0 \vdash^{\mathit{simp}} \pi \rightsquigarrow \mathit{fail}}$$

where $\Phi_1 = \Phi_0[\pi, \pi_0]$, $\Phi_1(\pi_0).I = (\eta(\pi) = 1, \infty)$. We have that:

$$\frac{\{(C\,(T^2\,a_2), \pi_0)\} = \mathit{matches}(C\,(T\,a_1), \Theta) \quad \Theta, \Phi_2 \vdash^{\mathit{simp}} (C\,(T^2\,a_2)) \rightsquigarrow \mathit{fail}}{\Theta, \Phi_1 \vdash^{\mathit{simp}} (C\,(T\,a_1)) \rightsquigarrow \mathit{fail}}$$

where $\Phi_2 = \Phi_1[C\,(T\,a_1), \pi_0] = \ \mathit{fail}$ because $\eta(C\,(T\,a_1)) \not< \Phi_1(\pi_0).I.v_1 = 1$.

By rule ($\mathtt{R}_0$), we have that $\Theta \vdash^{\mathtt{simp}_0} \{D\,\mathtt{I},\ C\,a\} \rightsquigarrow \{C\,a\}$, meaning that $D\,\mathtt{I}$ can be removed and $C\,a$ cannot be further reduced.

The following theorem states the correctness of the constraint simplification function defined in Figure 5.

Theorem 3 (Correctness of $\vdash^{\mathtt{simp}}$) *If $\Theta, \Phi \vdash^{\mathit{simp}} P \rightsquigarrow Q$ holds, then $\Theta, Q \Vdash P$ is provable and Q cannot be further simplified, i.e. $\Theta, \Phi \vdash^{\mathit{simp}} Q \rightsquigarrow Q$.*

Proof: Induction over $\Theta, \Phi \vdash^{\mathtt{simp}} P \rightsquigarrow Q$. □

6 Conclusion

This paper presents a termination criterion and terminating algorithms for constraint simplification and improvement, based on the use of a value that always decreases on each recursive step in these algorithms. The termination criterion defined can be used in any form of constraint simplification and improvement algorithm during type inference.

The use of this criterion eliminates the need for imposing syntactic conditions on Haskell type class and instance declarations and the need for using a recursion stack depth limit in order to guarantee termination of type inference in the presence of multi-parameter type classes, in case these syntactic conditions are chosen by programmers not to be enforced.

Since type class constraint satisfiability is in general undecidable, there exist instances of this problem for which the algorithm presented in this paper incorrectly reports unsatisfiability. However, practical examples where this occurs are expected to be very rare. The algorithms have been implemented and used in a prototype front-end for Haskell, available at

`http://github.com/rodrigogribeiro/mptc`

For all examples mentioned in the literature, Haskell libraries that use multi-parameter type classes and tests used by the Haskell GHC compiler, involving all pertinent GHC extensions, the algorithm works as expected without the need for any compilation flag.

In comparison to the use of a recursion depth limit, our approach has the advantage that type-correctness is not implementation dependent (a constraint is or is not satisfiable with respect to a given set of constraint axioms). The use of a recursion depth limit can make a constraint set satisfiable in one implementation and unsatisfiable in another that uses a lower limit. Incorrectly reporting unsatisfiability can occur in both cases, but is expected to be extremely rare with our approach. We are not aware of any practical example where this occurs.

The main disadvantages of our approach are that it is not syntactically possible to characterize such incorrect unsatisfiability cases and it is not very easy for programmers to understand how type class constraints are handled in such a case, if and when it occurs.

Acknowledgements We would like to thank the anonymous reviewers for their careful work, which has been very useful to improve the paper.

References

1. A. Gill: MTL — The Monad Transformer Library. `http://hackage.haskell.org/package/mtl` (2006)
2. Baader, F., Snyder, W.: Unification theory. In: J. Robinson, A. Voronkov (eds.) Handbook of Automated Reasoning, vol. 1, pp. 447–533. Elsevier Science Publishers (2001)
3. C. Camarão, L. Figueiredo and C. Vasconcellos: Constraint-set Satisfiability for Overloading. In: Proc. of the 6th ACM SIGPLAN International Conf. on Principles and Practice of Declarative Programming (PPDP'04), pp. 67–77 (2004)
4. C. Camarão, R. Ribeiro, L. Figueiredo and C. Vasconcellos: A Solution to Haskell's Multi-Parameter Type Class Dilemma. In: Proc. of the 13th Brazilian Symposium on Programming Languages (SBLP'2009), pp. 5–18 (2009). `http://www.dcc.ufmg.br/~camarao/CT/solution-to-mptc-dilemma.pdf`
5. C. Hall, K. Hammond, S. P. Jones and P. Wadler: Type Classes in Haskell. ACM Transactions on Programming Languages and Systems **18**(2), 109–138 (1996)
6. G. Smith: Polymorphic type inference for languages with overloading and subtyping. Ph.D. thesis, Cornell Univ. (1991)
7. Jones, M., Diatchki, I.: Language and Program Design for Functional Dependencies. In: ACM SIGPLAN Haskell Workshop, pp. 87–98 (2008)
8. Jones, S.P., et al.: The Haskell 98 Language and Libraries: The Revised Report. Journal of Functional Programming **13**(1), 0–255 (2003). `http://www.haskell.org/definition/`
9. L. Zhao: Solving and Creating Difficult Instances of Posts Correspondence Problem. Master's thesis, Department of Computer Science, University of Alberta (2002)
10. M. Chakravarty, G. Keller and S. P. Jones: Associated type synonyms. In: Proc. of the 10th ACM SIGPLAN International Conf. on Functional Programming (ICFP'05), pp. 241–253 (2005)
11. M. Chakravarty, G. Keller, S. P. Jones and S. Marlow: Associated types with class. In: Proc. of the ACM Symp. on Principles of Prog. Languages (POPL'05), pp. 1–13 (2005)
12. M. Jones: Qualified Types. Cambridge University Press (1994)
13. M. Jones: Simplifying and Improving Qualified Types. In: Proc. of the ACM Conf. on Functional Prog. and Comp. Architecture (FPCA'95), pp. 160–169 (1995)
14. M. Jones: Type Classes with Functional Dependencies. In: Proc. of the European Symp. on Programming (ESOP'2000) (2000). LNCS 1782
15. M. Sulzmann, G. Duck, S. P. Jones and P. Stuckey: Understanding Functional Dependencies via Constraint Handling Rules. Journal of Functional Programming **17**(1), 83–129 (2007)
16. Milner, R.: A theory of type polymorphism in programming. Journal of Computer and System Sciences **17**, 348–375 (1978)
17. P. Stuckey and M. Sulzmann: A Theory of Overloading. ACM Trans. on Prog. Lang. and Systems (TOPLAS) **27**(6), 1216–1269 (2005)
18. P. Wadler and S. Blott: How to make *ad-hoc* polymorphism less *ad hoc*. In: Proc. of the 16 $^{\text{th}}$ ACM Symp. on Principles of Prog. Lang. (POPL'89), pp. 60–76. ACM Press (1989)
19. S. P. Jones and others: GHC — The Glasgow Haskell Compiler. `http://www.haskell.org/ghc/` (1998)
20. S. P. Jones and others: GHC — The Glasgow Haskell Compiler 7.0.4 User's Manual. `http://www.haskell.org/ghc/` (2011)