Constraints in SGGS

Maria Paola Bonacina*
Dipartimento di Informatica
Università degli Studi di Verona
Verona, I-37134, Italy
E-mail: mariapaola.bonacina@univr.it

David A. Plaisted
Department of Computer Science
UNC Chapel Hill
Chapel Hill, NC 27599-3175, USA
E-mail: plaisted@cs.unc.edu

June 12, 2014

Abstract

We discuss the constraint system in the SGGS inference system, which stands for semantically-guided goal-sensitive theorem proving.

*Research supported in part by grant no. 2007-9E5KM8 of the Ministero dell’Istruzione Università e Ricerca, Italy, and by COST Action IC0901 of the European Union.
1 Basic definitions and concepts for SGGS

1.1 Constrained clauses

The SGGS inference system takes as input

- a set $S$ of clauses,
- an initial interpretation $I$, and
- an ordering $\prec$ on ground literals,
and builds a sequence of clauses that represents a partial model of $S$.

While in propositional logic a partial model of a set of clauses can be represented by a sequence of literals, in first-order logic it needs a sequence of clauses with constraints:

**Definition 1.1 (Constraint)** An atomic constraint is either

1. empty, denoted by true, or

2. an expression of the form $x \equiv y$ or $\text{top}(t) = f$, where

   (a) $x$ and $y$ are variables,
   
   (b) $f$ is a function symbol, and
   
   (c) $t$ is a term.

A constraint is either

1. an atomic constraint, or

2. the negation, conjunction, or disjunction of constraints.

The meaning of the constraints is defined by

1. $\models t \equiv u$ for ground terms $t$ and $u$ if $t$ and $u$ are the same element of the Herbrand universe.

2. $\models \text{top}(t) = f$ if the top symbol of ground term $t$ is $f$. 
Definition 1.2 (Standard form) A constraint is in standard form, if it is a conjunction of distinct atomic constraints of the form \( x \not\equiv y \) and \( \text{top}(x) \neq f \), where \( x \) and \( y \) are variables.

- A constraint \( \text{top}(x) \neq f \) says that \( x \) cannot be replaced by a term whose top function symbol is \( f \), while
- a constraint \( x \not\equiv y \) specifies that \( x \) and \( y \) may not be replaced by identical terms.

Definition 1.3 (Constrained clause) A constrained clause is a formula \( A \sqsupset C \), where

- \( A \) is a constraint and
- \( C \) is a clause.

Any variable that appears in \( A \) and not in \( C \) is implicitly existentially quantified.

In a constrained clause \( A \sqsupset C \) a literal \( L \) may be selected, written \( A \sqsupset C[L] \).

- By analogy, \( A \sqsupset L \) is called a constrained literal,
- and by convention, if \( L \) is the selected literal of \( C \), and \( C' \equiv C\vartheta \), then \( L' \equiv L\vartheta \) is the selected literal of \( C' \).
- \( \text{true} \sqsupset C \) is usually abbreviated as \( C \).
Definition 1.4 (Constrained ground instances) Given a constrained clause \( A \triangleright C \) its set of constrained ground instances (cgi) is

\[
Gr(A \triangleright C) = \{ C\vartheta : \models A\vartheta, \text{ C\vartheta ground.}\}
\]

Note how

- \( Gr(false \triangleright C) = \emptyset \), while
- \( Gr(true \triangleright C) \) contains all ground instances of \( C \).

The same notion applies to a single literal:

\[
Gr(A \triangleright L) = \{ L\vartheta : \models A\vartheta,\text{ L\vartheta ground}\}.
\]

For a single literal \( \neg Gr(A \triangleright L) \) or \( Gr(A \triangleright \neg L) \) is the set

\[
\{ \neg L' : L' \in Gr(A \triangleright L)\}.
\]

Example 1.1 For a clause \( x \not\equiv y \triangleright P(x, y) \),

1. \( P(a, b) \in Gr(x \not\equiv y \triangleright P(x, y)) \),
2. \( P(b, b) \not\in Gr(x \not\equiv y \triangleright P(x, y)) \).

Definition 1.5 The minimal constrained ground instance of a constrained literal \( A \triangleright L \) is

\[
cmin(A \triangleright L) = \begin{cases} 
\min\{M : M \in Gr(A \triangleright L)\} & \text{if } Gr(A \triangleright L) \neq \emptyset, \\
M_\infty & \text{otherwise.}
\end{cases}
\]
where the ordering $\prec$ is suitably defined.

The minimal constrained ground instance of a constrained clause $A \triangleright C[L]$ is the minimal constrained ground instance of its selected literal:

$$cmin(A \triangleright C[L]) = cmin(A \triangleright L).$$

### 1.2 Clause Sequences

SGGS works with clause sequences that satisfy certain requirements, which will be omitted here.

### 2 Intersection, partition, splitting and difference

**Definition 2.1** Constrained literals $A \triangleright L$ and $B \triangleright M$

1. intersect if $at(Gr(A \triangleright L)) \cap at(Gr(B \triangleright M)) \neq \emptyset$, and

2. are disjoint, otherwise.

Intersection does not require that two literals have the same sign, because it is defined based on the atoms of their constrained ground instances.
Definition 2.2 (Partition) A partition of $A \triangleright C \langle L \rangle$, where $A$ is satisfiable, is a set

$$\{A_i \triangleright C_i \langle L_i \rangle\}_{i=1}^n$$

such that

1. $Gr(A \triangleright C) = \bigcup_{i=1}^n \{Gr(A_i \triangleright C_i \langle L_i \rangle)\}$,

2. the constrained literals $A_i \triangleright L_i$ are pairwise disjoint,

3. all $A_i$’s are satisfiable, and

4. the $L_i$’s are chosen consistently with $L$.

Example 2.1 The set

$$\{\text{true} \triangleright P(f(z), y), \text{top}(x) \neq f \triangleright P(x, y)\}$$

is a partition of $\text{true} \triangleright P(x, y)$.

If $L$ and $M$ intersect, it is possible to split $A \triangleright C \langle L \rangle$ by $B \triangleright D[M]$:

Definition 2.3 (Splitting and difference) A splitting of $A \triangleright C \langle L \rangle$ by $B \triangleright D[M]$, denoted $\text{split}(C, D)$, is a partition $\{A_i \triangleright C_i \langle L_i \rangle\}_{i=1}^n$ of $A \triangleright C \langle L \rangle$ such that:

1. $\exists j, 1 \leq j \leq n$, such that $at(Gr(A_j \triangleright L_j)) \subseteq at(Gr(B \triangleright M))$, and

2. $\forall i, 1 \leq i \neq j \leq n$, $at(Gr(A_i \triangleright L_i))$ and $at(Gr(B \triangleright M))$ are disjoint;
and the difference $C - D$ is $\text{split}(C, D)$ with $C_j$ removed.

Clause $C_j$ is the representative of $D$ in $\text{split}(C, D)$.

SGGS needs to compute splitting and differences.

Computing $\text{split}(C, D)$ and $C - D$ introduces constraints, including non-standard ones, even when $C$ and $D$ have empty constraints to begin with:

**Example 2.2** A splitting of $true \triangleright P(x, y)$ by $true \triangleright P(f(w), g(z))$ is

- \{true $\triangleright P(f(w), g(z))$,
- $\text{top}(x) \neq f \triangleright P(x, y)$,
- $\text{top}(y) \neq g \triangleright P(f(x), y)\}$.

3 Constraints

In this section we present rules that manipulate constraints to compute clause differences and splittings, and standardize constraints.

These rules are sound, in the sense that premise and conclusion represent the same set of constrained ground instances.

If a conclusion is made of multiple clauses, it is read as their disjunction:

- if a rule has premise $A \triangleright C$ and conclusion $A_1 \triangleright C_1, \ldots, A_n \triangleright C_n$, then $Gr(A \triangleright C) = \bigcup_{i=1}^{n} Gr(A_i \triangleright C_i)$;
• if the conclusion is $\bot$, it means that $A$ is unsatisfiable.

3.1 Rules for constraints

In general we define $Gr(C - D)$ by

$$Gr(C - D) = \bigcup_{i=1, i \neq j}^{n} Gr(C_i)$$

for $\text{split}(C, D) = \{ A_i \triangleright C_i(L_i) \}_{i=1}^{n}$ and $C_j$ the representative of $D$.

According to Definition 2.3, given $A \triangleright C[L]$ and $B \triangleright D[M]$,

• if $at(L)$ and $at(M)$ do not unify, then

$$Gr(C - D) = Gr(C)$$

• If $at(L)$ and $at(M)$ unify, with $\sigma = mgu(at(L), at(M))$, then

$$\text{split}(C, D) = (C - D) \cup \{ A\sigma \land B\sigma \triangleright C[L]\sigma \}$$

and

$$(C - D) = (C - (A\sigma \land B\sigma \triangleright C[L]\sigma)).$$

Thus,

• if we have a way to compute $C - D$, we also have a way to compute $\text{split}(C, D)$, and

• we can restrict ourselves to compute $C - D$ under the assumption that $D$ is an instance $C\sigma$ of $C$. 
Definition 3.1 (Rules for clause difference) Given clauses $A \triangleright C$ and $B \triangleright D$, such that $D \equiv C\sigma$, the rules for clause difference are:

- If $\{x \leftarrow f(x_1, \ldots, x_n)\} \subseteq \sigma$ for some $x \in \text{vars}(C)$ and new variables $x_i$, $1 \leq i \leq n$, the DiffSim rule
  
  - applies $\{x \leftarrow f(x_1, \ldots, x_n)\}$ to make $C$ closer to being similar to $D$ and
  
  - on the other hand adds $\text{top}(x) \neq f$ to make the clauses disjoint:

  \[
  (A \triangleright C') - (B \triangleright D)
  \]
  
  \[
  (A \triangleright C')\{x \leftarrow f(x_1, \ldots, x_n)\} - (B \triangleright D), \quad A \land (\text{top}(x) \neq f) \triangleright C
  \]

- If $C$ and $D$ are similar, which means $\sigma$ only replaces variables by variables, and $\{x \leftarrow y\} \subseteq \sigma$ for distinct variables $x, y \in \text{vars}(C)$, the DiffVar rule
  
  - applies $\{x \leftarrow y\}$ to make $C$ closer to a variant of $D$ and
  
  - on the other hand adds $x \not\equiv y$ to make the clauses disjoint:

  \[
  (A \triangleright C') - (B \triangleright D)
  \]
  
  \[
  (A \triangleright C')\{x \leftarrow y\} - (B \triangleright D), \quad (x \neq y \land A) \triangleright C
  \]

- If $C$ and $D$ are variants but not identical, the DiffId rule
makes them identical:

\[
(A \triangleright C) - (B \triangleright D) = (A \triangleright C)\sigma - (B \triangleright D)
\]

- The DiffElim rule replaces difference by negation if \(C\) and \(D\) are identical:

\[
(A \triangleright C) - (B \triangleright C) = (A \wedge \neg B) \triangleright C
\]

Since \(B\) is a conjunction of constraints, \(\neg B\) is a disjunction of their negations.

Thus, the system needs rules that restore disjunctive normal form (DNF):

**Definition 3.2 (Rules for connectives)** The rules for connectives are:

- The Equiv rule replaces a constraint \(A\) by its disjunctive normal form \(\text{dnf}(A)\):

\[
\frac{A \triangleright C}{\text{dnf}(A) \triangleright C}
\]

- The Div rule subdivides disjunction:

\[
\frac{(A \lor B) \triangleright C}{A \triangleright C, B \triangleright C}
\]
Next come rules that reduce identity constraints to standard form.

For these rules we can assume that a constraint is a conjunction of atomic constraints and their negations.

**Definition 3.3 (Rules for identity)** The rules for identity are:

- **The ElimId1 rule** eliminates a constraint between variable and term: if $x \not\in \text{vars}(s)$, then:

  $$
  \frac{(A \land x \equiv s) \triangleright C}{(A \triangleright C)\{x \leftarrow s\}}
  $$

  if $x \in \text{vars}(s)$ and $s$ is not a variable, then:

  $$
  \frac{(A \land x \equiv s) \triangleright C}{\bot}
  \quad \frac{(A \land x \not\equiv s) \triangleright C}{(A \triangleright C)}
  $$

- **The ElimId2 rule** detects a conflict: if $f \neq g$, $m \geq 0$, $n \geq 0$, then:

  $$
  \frac{(A \land f(s_1, \ldots, s_n) \equiv g(t_1, \ldots, t_m)) \triangleright C}{\bot}
  $$

- **The ElimId3 rule** eliminates a satisfied constraint: if $f \neq g$, $m \geq 0$, $n \geq 0$, then:

  $$
  \frac{(A \land f(s_1, \ldots, s_n) \not\equiv g(t_1, \ldots, t_m)) \triangleright C}{A \triangleright C'}
  $$

- **The ElimId4 rule** decomposes an identity: if $n \geq 0$, then:

  $$
  \frac{(A \land f(s_1, \ldots, s_n) \equiv f(t_1, \ldots, t_n)) \triangleright C}{(A \land s_1 \equiv t_1 \land \ldots \land s_n \equiv t_n) \triangleright C'}
  $$
• The ElimId5 rule decomposes a negated identity: if \( n \geq 0 \), then:

\[
\begin{align*}
(A \land f(s_1, \ldots, s_n) \not\equiv f(t_1, \ldots, t_n)) & \triangleright C \\
(A \land (s_1 \not\equiv t_1 \lor \ldots \lor s_n \not\equiv t_n)) & \triangleright C
\end{align*}
\]

After this rule, of course, the constraint can be reduced to dnf and split into conjuncts as before.

• The ElimId6 rule eliminates a negated identity between variable and non-variable term:

\[
\begin{align*}
(A \land x \not\equiv f(s_1, \ldots, s_n)) & \triangleright C \\
A \land \text{top}(x) \not\equiv f & \triangleright C, \quad ((A \land f(s_1, \ldots, s_n) \not\equiv f(y_1, \ldots, y_n)) \triangleright C)\rho
\end{align*}
\]

where

- \( \rho = \{x \leftarrow f(y_1, \ldots, y_n)\} \),
- \( n \geq 0 \), and
- for all \( i, 1 \leq i \leq n \), \( y_i \) is a new variable;
- (this in turn permits an application of ElimId5)

• The ElimId7 rule detects a conflict: if \( s \) is a variable or constant, then:

\[
\begin{align*}
(A \land s \not\equiv s) & \triangleright C \\
\bot
\end{align*}
\]

The ElimId5 rule also calls for restoration of DNF.

The rules for top symbol eliminate all top symbol constraints, except those in standard form \( \text{top}(x) \not\equiv f \):
Definition 3.4 (Rules for top symbol) The rules for top symbol are

- **The ElimTop1 rule** detects a conflict in a positive constraint: if \( f \neq g, n \geq 0 \), then:

\[
\frac{A \land \text{top}(f(s_1, \ldots, s_n)) = g \triangleright C}{\bot}
\]

- **The ElimTop2 rule** eliminates a satisfied positive constraint: if \( n \geq 0 \), then:

\[
\frac{A \land \text{top}(f(s_1, \ldots, s_n)) = f \triangleright C}{A \triangleright C}
\]

- **The ElimTop3 rule** eliminates a satisfied negative constraint: if \( f \neq g, n \geq 0 \), then:

\[
\frac{A \land \text{top}(f(s_1, \ldots, s_n)) \neq g \triangleright C}{A \triangleright C}
\]

- **The ElimTop4 rule** detects a conflict in a negated constraint: if \( n \geq 0 \), then:

\[
\frac{A \land \text{top}(f(s_1, \ldots, s_n)) \neq f \triangleright C}{\bot}
\]

- **The ElimTop5 rule** eliminates a positive constraint: if \( n \geq 0 \), then:

\[
\frac{A \land \text{top}(x) = f \triangleright C}{(A \triangleright C) \{x \leftarrow f(x_1, \ldots, x_n)\}}
\]

where for all \( i, 1 \leq i \leq n \), \( x_i \) is a new variable.
The combined effect of all rules is to standardize all constraints (cf. Definition 1.2).

However, the application of the identity rules may not terminate:

**Example 3.1** Consider a clause \((x \neq f(y) \land y \neq f(x)) \triangleright P(x,y))\):

By applying the ElimId6 rule one gets the two clauses

1. \((\text{top}(x) \neq f \land y \neq f(x)) \triangleright P(x,y)\) and

2. \((f(z) \neq f(y) \land y \neq f(f(z)) \triangleright P(f(z), y))\).

Using ElimId5, the latter clause becomes

\[(z \neq y \land y \neq f(f(z)) \triangleright P(f(z), y)),\]

which then by another application of ElimId6, yields the two clauses

1. \((z \neq y \land \text{top}(y) \neq f) \triangleright P(f(z), y))\) and

2. \((z \neq f(w) \land f(w) \neq f(f(z)) \triangleright P(f(z), f(w)))\).

Using ElimId5 again, the latter clause becomes

\[(z \neq f(w) \land w \neq f(z) \triangleright P(f(z), f(w))),\]

whose constraint is a variant of the original one.

SGGs does not need that every series of applications of these rules terminate.

It suffices to show that the computation of clause difference terminates:
Theorem 3.1 Given \( A \triangleright C \) and \( B \triangleright D \), such that \( D \equiv C\sigma \), and \( A \) and \( B \) are in standard form, any application of the clause difference rules to \( C - D \), where

1. any application of DiffElim or ElimId5 is followed by conversion to DNF, and
2. all constraints are restored to standard form after every application of a clause difference rule,

is guaranteed to terminate.

Proof: First we show that the rules for clause difference do not cause non-termination.

1. DiffId and DiffElim can be applied only once.
2. DiffVar can be applied only a finite number of times, because each application decreases the number of variables in \( C \).
3. Each DiffSim step applies to \( C \) a substitution \( \{ x \leftarrow f(x_1, \ldots x_n) \} \) from \( \sigma \): since \( \sigma \) contains finitely many such pairs, DiffSim can be applied only a finite number of times.

Then we prove that standardization between an application of a clause difference rule and the next is guaranteed to terminate:

1. DiffId only renames variables, which does not enable any other rule.
2. DiffVar adds an \( x \not= y \), which is in standard form, and applies a substitution \( \{ x \leftarrow y \} \), whose only effect may be to replace an \( x \not= y \) by an \( x \not= x \), eliminated by ElimId7.

3. DiffSim adds a \( \text{top}(x) \not= f \), which is in standard form, and applies a substitution \( \{ x \leftarrow f(x_1, \ldots, x_n) \} \), which may have two effects.

   - One is to replace the occurrence of \( x \) in a constraint \( \text{top}(x) \not= g \) by \( f(x_1, \ldots, x_n) \).
     This enables either ElimTop3 or ElimTop4, which terminate.

   - The other is to transform an \( x \not= y \) into an \( f(x_1, \ldots, x_n) \not= y \), enabling ElimId6.

ElimId6 adds a \( \text{top}(x) \not= f \), which is in standard form, and applies another substitution of the same form, so that eventually a subset of the variables may be replaced by terms \( f(x_1, \ldots, x_n) \) where the \( x_i \)'s are new.

   - This can only be done a finite number of times, because the new variables will never be replaced in this way.

   - If two such substitutions are applied to a \( z \not= w \), an \( f(x_1, \ldots, x_n) \not= f(y_1, \ldots, y_n) \) may arise.
   ElimId5 applies to such a constraint, followed by conversion to DNF.

   - The result is a disjunction of constrained clauses, each containing in its constraint an \( x_i \not= y_i \), for some
\( i \), which is in standard form.

4. DiffElim yields \((A \land \neg B) \triangleright C\), followed by conversion to DNF.

The effect may be to add \( x \equiv y \) (negation of \( x \not\equiv y \) in \( B \)) or \( \text{top}(x) = f \) (negation of \( \text{top}(x) \neq f \) in \( B \)).

- In the first case, ElimId1 applies \( \{x \leftarrow y\} \), covered in Case (2) of this proof.
- In the second case, ElimTop5 applies \( \{x \leftarrow f(x_1, \ldots, x_n)\} \), covered in Case (3) of this proof.

The set of inference rules for constraints is completed by rules that remove from a clause \( A \triangleright C \) variables that appear in \( A \) but not in \( C \).

These rules do not affect Theorem 3.1.

**Definition 3.5 (Rules for variable removal)** Given a clause \( A \triangleright C \), such that

- \( A \) is in standard form,
- \( y \in \text{vars}(A) \), and
- \( y \not\in \text{vars}(C) \),

the rules for variable removal are:
The ElimVar1 rule detects that the constraints on \( y \) are satisfiable:

\[
\text{if } \exists f \in \text{fun}(S), \text{ such that } \text{ar}(f) \geq 1 \text{ and } \text{top}(y) \neq f \notin A, \text{ then:}
\]

\[
\frac{A \triangleright C}{\text{Rem}(y, A) \triangleright C}
\]

where \( \text{Rem}(y, A) \) is \( A \) with all conjuncts of the form

\[
- \text{top}(y) \neq g, g \neq f, \text{ and}
\]

\[
- y \neq z, \text{ where } z \text{ is another variable},
\]

replaced by true;

The ElimVar2 rule detects that the constraints on \( y \) are unsatisfiable:

\[
\text{if } \forall f \in \text{fun}(S), A \text{ contains a constraint } \text{top}(y) \neq f, \text{ then:}
\]

\[
\frac{A \triangleright C}{\bot}
\]

The ElimVar3 rule removes \( y \) by replacing it with all possible constants:

\[
\text{if } \forall f \in \text{fun}(S) \text{ such that } \text{ar}(f) \geq 1, A \text{ contains a constraint } \text{top}(y) \neq f, \text{ then:}
\]

\[
\frac{A \triangleright C}{(\bigvee_{c \in \text{fun}(S), \text{ar}(c) = 0} A\{y \leftarrow c\}) \triangleright C}
\]

To justify these rules,
• If the conditions of the ElimVar1 rule are met, all constraints about \( y \) can be satisfied by replacing \( y \) with a term having \( f \) as top symbol. Since there are infinitely many such terms, one can always be chosen to satisfy the constraints of the form \( y \not\equiv z \).

• The ElimVar2 rule deals with the case in which all function and constant symbols are prohibited for \( y \), which means that the constraint is unsatisfiable.

• The ElimVar3 rule deals with the case in which all function symbols (i.e., having arity one or more) are prohibited for \( y \); in this case,

  – \( y \) has to be replaced by a constant symbol, and
  – since there are only finitely many of them, \( A \) can be replaced by a disjunction of constraints.

Also ElimVar3 relies on subsequent conversion to DNF.

It is possible to test whether a constraint \( A \) is satisfiable, by applying the rules in this section to \( A \triangleright false \).

• If the result is \( false \), then \( A \) is satisfiable; if the result is \( \bot \), then \( A \) is unsatisfiable.

• Since \( A \) is valid if and only if \( \neg A \) is unsatisfiable, one can test the validity of \( A \) by testing \( \neg A \) for satisfiability.
3.2 Computing minimal constrained ground instances

It is helpful at times to compute $cmin$. In this section we cover the issue of how to compute

$$cmin(A \triangleright L),$$

assuming that $A$ is in standard form.

- If $A$ is unsatisfiable, $Gr(A \triangleright L) = \emptyset$ and
  $$cmin(A \triangleright L) = M_\infty$$
  where $M_\infty$ represents infinity.

- If $A$ is satisfiable, the idea is to compute a finite set of constrained literals
  $$\mathcal{T} = \{A\alpha \triangleright L\alpha\},$$
  and then consider those $L\alpha$ such that $A\alpha$ is satisfied.

The literal $cmin(A \triangleright L)$ will be the smallest of these $L\alpha$ in the ordering $\prec$.

The set $\mathcal{T}$ is initialized to contain $A \triangleright L$ itself and the candidate for $cmin$ is set to $M_\infty$.

3.2.1 First Phase

In a first phase, for each constraint $top(x) \neq f$ in $A$, $\mathcal{T}$ is expanded to specify all function symbols other than $f$ as the top symbol for $x$. 
This is done by adding the instances
\[ \{ A' \triangleright L \vartheta : g \in \text{fun}(S), \text{ar}(g) = k, g \neq f, \vartheta = \{ x \leftarrow g(y_1, \ldots, y_k) \} \} \],

where

- \( A' \) is \( A \) with \( \text{top}(x) \neq f \) removed, and
- \( \forall i, 1 \leq i \leq k, y_i \) is new.

If \( A \) contains at least one constraint \( \text{top}(x) \neq f \), the original constrained literal \( A \triangleright L \) can be removed from \( \mathcal{T} \) after this expansion.

- The result of repeatedly applying this rule is a set \( \mathcal{T} \) of constrained literals with no constraint of the form \( \text{top}(x) \neq f \).
- If \( A \) originally contained at least one constraint \( \text{top}(x) \neq f \), the constraints in \( \mathcal{T} \) are no longer in standard form: they are conjunctions of constraints of the form \( s \not\equiv t \) for terms \( s \) and \( t \).

The rules in Definition 3.3 can be applied to transform them into standard form.

- Since unrestricted application of the rules in Definition 3.3 is not guaranteed to terminate,
  - this simplification phase can be applied only with a bound on the number of rule applications, and
there is no guarantee in general to reach a set with constraints in standard form.

However, maintaining all constraints in standard form is not necessary to compute \(cmin(A \triangleright L)\).

### 3.2.2 Second Phase

A second phase interleaves variable instantiation, bounded simplification by the rules in Definition 3.3, constraint testing, and discovery of \(cmin(A \triangleright L)\).

- For variable instantiation, the idea is to instantiate each variable to all possible top symbols.

  Thus if \(x \in \text{vars}(A\alpha)\) for some \(A\alpha \triangleright L\alpha\) in \(\mathcal{T}\), \(A\alpha \triangleright L\alpha\) is replaced by \(A\alpha\vartheta \triangleright L\alpha\vartheta\), where

  - \(\vartheta = \{x \leftarrow g(y_1, \ldots, y_k)\}\),
  - \(g \in \text{fun}(S)\),
  - \(\text{ar}(g) = k\), and
  - \(\forall i, 1 \leq i \leq k, y_i\) is new.

- For constraint testing, any \(A\alpha \triangleright L\alpha\) such that \(A\alpha\) is unsatisfiable is removed from \(\mathcal{T}\).

- For discovery of \(cmin(A \triangleright L)\), any \(A\alpha \triangleright L\alpha \in \mathcal{T}\) such that \(A\alpha\sigma\) simplifies to \textit{true}, where
– $\sigma$ is a substitution that replaces all variables of $A\alpha \triangleright L\alpha$ by constant symbols,

yields a candidate $L\alpha \sigma$ for $cmin(A \triangleright L)$.

Eventually at least one such candidate literal $M$ will be found, because the original constraint $A$ is satisfiable.

• Any $A\alpha \triangleright L\alpha \in T$ such that $L\alpha \triangleright M$ can be deleted from $T$, even if $L\alpha$ contains variables, because $\prec$ extends the size ordering.

• Constrained literals $A\alpha \triangleright L\alpha$ in $T$ such that $L\alpha \prec M$, are retained for further variable instantiation and constraint testing.

• If a ground literal $M'$ such that $M' \prec M$ is produced, $M$ is deleted, and $M'$ replaces it as current candidate for $cmin(A \triangleright L)$.

This procedure terminates when $T$ is a singleton, and its only element is $cmin(A \triangleright L)$.

• This is guaranteed to happen, because $A$ is satisfiable, $\triangleright$ is well-founded, and variable instantiation causes the literals $L\alpha$ in $T$ to grow in size, and therefore in the ordering $\prec$.

• This procedure works because the literals $L\alpha$ in $T$ become larger and larger in $\prec$. 