Chapter 7

Proof producing CLP(LI+UIF)

CLP(Q) [52] is a useful building block for verification tools. However, CLP(Q) does not currently produce proofs, which are needed to compute interpolants, and does not deal with the theory of uninterpreted functions, which is useful for modelling complex operations when verifying programs. In this chapter, we present a tool CLP(LI+UIF) that checks unsatisfiability of conjunctive constraints in the theory of linear arithmetic and uninterpreted functions and also produces a proof tree when unsatisfiability is detected.

The existing simplex based proof producing algorithms [14, 23] use a version of simplex that does not apply constant propagation. These algorithms construct proofs by relying on an instrumentation of the input constraints. This instrumentation leads to the creation of many additional variables. In this chapter, we present an alternative proof producing simplex based algorithm that relies on an instrumentation of an incremental, constant propagating simplex. Our instrumentation does not require incremental simplex to introduce additional variables for proof construction and does not prohibit constant propagation.

In following sections, first we will present the algorithm used in CLP(Q) and its extension for supporting uninterpreted functions. Second, we will discuss the incompatibility of existing algorithms for proof tree generation with CLP(Q). Third, we will present our instrumentation of the algorithm in CLP(Q).

7.1 CLP(Q)

CLP(Q) [52] is a linear programming tool. Since, We are only interested in the unsatisfiability of conjunctive constraints, we will only consider *phase 1* of simplex. In CLP(Q), this phase is implemented as a version of incremental simplex.

The incremental simplex takes as input a sequence of linear atoms. At any instant, the input so far is stored in a so called *solved form* that represents the input in a normal form. Given the next input from the input sequence and the current solved form, the incremental simplex computes the next solved form. A solved form of a conjunctive constraint exists if and only if the conjunctive constraint is satisfiable. Therefore, failure to compute a solved form indicates that the input considered so far is unsatisfiable. In practice, the incremental simplex is more efficient than a non-incremental one for satisfiablity checking [53].

The algorithm of the CLP(Q) solver is described in [51] and is an optimized version of algorithm presented in [70]. We will now reformulate this algorithm in the notation that is convenient to us. The CLP(Q) solver has the following important optimizations that affect proof tree extraction.

- The CLP(Q) solver avoids introduction of as many slack variables as possible.
- If the input implies that a variable is equal to a constant then the CLP(Q) solver replaces this variable with the constant.

Solved form

In general, a solved form is a variant of the standard simplex tableau [77]. There are various kinds of solved forms with different properties regarding data structure representation, ability to detect equalities and disequalities, and efficiency of transforming a conjunctive constraints into the solved form [53]. The CLP(Q) uses a solved form in which equality detection is most efficient and therefore the treatment of disequalities is trivial, but the cost of computing the solved form is high. Equality detection is also very significant for us since congruence checker for uninterpreted functions depends on equality detection.

Formally, the solved form in CLP(Q) is a tuple $(X, \mathit{Basis}, \mathit{Def}, \mathit{Low}, \mathit{Up}, \mathit{Active}, \mathit{Val})$ where

- $X = \{x_1, \ldots, x_n\}$ is a finite ordered set of rational variables,
- $\mathit{Basis} \subseteq X$ is called a *basis*,
- $\mathit{Def} : X \to$ linear terms, Def assigns definitions to variables and we require that for each $k \in 1..n$,

$$\mathit{Def}(x_k) = c + \sum_{j \in J} c_j x_j,$$

  where $c \in \mathbb{Q}$, $J \subseteq 1..n$ and $\forall j \in J.\ c_j \in \mathbb{Q} \setminus \{0\}$,
- $\mathit{Low} : X \to \mathbb{Q} \cup \{-\infty\}$, Low defines lower bounds on variables,
- $\mathit{Up} : X \to \mathbb{Q} \cup \{+\infty\}$, Up defines upper bounds on variables,
- $\mathit{Active} : X \to \{\texttt{none}, \texttt{lower}, \texttt{upper}\}$,
- $\mathit{Val} : X \to \mathbb{Q}$,
- and the conditions listed below are satisfied.

Let $k \in 1..n$. x_k is *undefined* if $\mathit{Def}(x_k) = x_k$ and is *defined* otherwise. x_k is *unbounded* if $\mathit{Low}(x_k) = -\infty$ and $\mathit{Up}(x_k) = +\infty$, otherwise x_k is *bounded.* x_k is *active* if $\mathit{Active}(x_k) \neq \texttt{none}$, and is *inactive* otherwise. A solved form must satisfy the following conditions:

(1) $\mathit{Def}(x_k)$ only contains undefined variables.

(2) If $x_k \in \mathit{Basis}$ then x_k is defined, bounded, inactive, and all variables appearing in $\mathit{Def}(x_k)$ are active.

(3) If $x_k \notin \mathit{Basis}$ and x_k is defined then x_k is unbounded and inactive.

(4) If x_k is active then x_k is bounded, undefined, and there is $x_b \in \mathit{Basis}$ such that x_k occurs in $\mathit{Def}(x_b)$.

(5) $\mathit{Low}(x_k) < \mathit{Up}(x_k)$.

(6) If $\mathit{Active}(x_k) = \texttt{lower}$ then $\mathit{Low}(x_k) \neq -\infty$, and if $\mathit{Active}(x_k) = \texttt{upper}$ then $\mathit{Up}(x_k) \neq +\infty$.

(7) If x_k is undefined then

$$\mathit{Val}(x_k) = \begin{cases} \mathit{Low}(x_k) & \text{if } \mathit{Active}(x_k) = \texttt{lower}, \\ \mathit{Up}(x_k) & \text{if } \mathit{Active}(x_k) = \texttt{upper}, \\ 0 & \text{if } \mathit{Active}(x_k) = \texttt{none}. \end{cases}$$

(8) If x_k is defined and $\mathit{Def}(x_k) = c + \sum_{j \in J} c_j x_j$ then $\mathit{Val}(x_k) = c + \sum_{j \in J} c_j \mathit{Val}(x_j)$.

(9) If $x_k \in \mathit{Basis}$ then $\mathit{Low}(x_k) \leq \mathit{Val}(x_k) \leq \mathit{Up}(x_k)$.

Kind	Condition	Basis	Defined	Active	Bounded	Remark
1	$x_k \in \mathit{Basis}$	✓	✓*	×*	✓*	If $\mathit{Def}(x_k) = c + \sum_{j \in J} c_j x_j$ then $\forall j \in J.\ x_j$ is undefined and active*
2	$x_k \notin \mathit{Basis}$ x_k is defined	×	✓	×*	×*	If $\mathit{Def}(x_k) = c + \sum_{j \in J} c_j x_j$ then $\forall j \in J.\ x_j$ is undefined*
3	x_k is active	×*	×*	✓	✓*	$\exists x_b \in \mathit{Basis}.\ x_k \in \mathit{Smb}(\mathit{Def}(x_b))$*
4	x_k is undefined x_k is inactive	×*	×	×		

Figure 7.1: Solved form implicitly induces above four kinds of variables. * denotes the property is a restriction imposed by a condition of solved form.

Conditions (1)–(4) impose a syntactic restriction, while (5)–(9) require arithmetic evaluation of solved form. *Def* represents a set of linear equations and condition (1) states that these equations are in a triangular form, which is usually obtained by Gaussian elimination. Conditions (2)–(4) induce four kinds of variables in the solved form that are presented in Figure 7.1. Note that variables of the fourth kind vacuously satisfy (2)–(4), since they violate the respective if-conditions.

A solved form is equivalent to the conjunctive constraint.

$$\bigwedge_{k=1}^{n} (\ x_k = \mathit{Def}(x_k) \wedge \mathit{Low}(x_k) \leq x_k \leq \mathit{Up}(x_k)\ ) \tag{7.1}$$

The conditions (1)–(9) imply that conjunctive constraints in Equation (7.1) are satisfiable. For a given solved form, we can construct a satisfying assignment Val' in following way. We choose assignments for variables in order of first kind, third kind, fourth kind, and second kind.

- For each x_k variable of first or third kind, let $\mathit{Val}'(x_k) = \mathit{Val}(x_k)$.
- Let x_k be a variable of fourth kind. x_k can only appear in the definition of variables of the second kind. The second kind variables are unbounded, therefore, we can choose any value for $\mathit{Val}'(x_k)$ that is between $\mathit{Low}(x_k)$ and $\mathit{Up}(x_k)$.
- Let x_k be a variable of the second kind such that $\mathit{Def}(x_k) = c + \sum_{j \in J} c_j x_j$. We have assigned Val' map for all the undefined variables therefore we can evaluate $\mathit{Def}(x_k)$ under assignments of Val'. Let $\mathit{Val}'(x_k) = c + \sum_{j \in J} c_j\, \mathit{Val}'(x_j)$.

Due to conditions (5)–(9), Val' is a satisfying assignment.

A satisfiable conjunctive constraint can always be transformed into an equisatisfiable solved form. The resulting solved form may contain more variables than the original constraint due to the introduction of slack variables in the process of transformation. The solved form may not be unique.

Example 3 (Solved form)**.** The constraints shown in Figure 7.2(a) are satisfiable. In figure 7.2(b), we show a solved form for the constraints. Variables x_1, x_2, x_3, and x_4 appear in the original constraints. Variables u, v, and w are slack variable that are introduced during the transformation to the solved form. x_4 and x_2 are variables of the first kind. x_3 is variable of the second kind. x_1, u, w, and v are variables of the third kind. There is no fourth kind of variable in this solved form therefore *Val* is satisfying assignment to the original constants.

CLP(Q) algorithm

Figures 7.3, 7.4, and 7.5 present incremental simplex in CLP(Q) [51]. This algorithm takes linear atoms as input sequence. Given an input and the current solved form, CLP(Q) computes the next solved form. If CLP(Q) fails to compute the next solved form then it throws an exception "`Unsatisfiable`".

If the input is an equation then AddEquality is called. If the input is an inequality then AddInequality is called. We refer to these two procedures as *entry* procedures.

$$\begin{aligned} x_1 - x_2 + 2x_3 - 2 &\le 0 \\ x_2 - x_1 - x_3 + 3 &\le 0 \\ x_4 + x_3 - 2 &\le 0 \\ 2 - x_4 &\le 0 \\ x_1 - 10 &\le 0 \\ 2 - x_2 &\le 0 \end{aligned}$$

(a)

	Variable	*Def*	*Low*	*Up*	*Active*	*Val*	
basis	x_4	$3+u+v-w$	2	$+\infty$	`none`	3	defined
basis	x_2	$-4+x_1-u-2v$	2	$+\infty$	`none`	6	defined
	x_3	$-1-u-v$	$-\infty$	$+\infty$	`none`	-1	defined
	x_1	x_1	$-\infty$	10	`upper`	10	active
slack variables	u	u	0	$+\infty$	`lower`	0	active
slack variables	v	v	0	$+\infty$	`lower`	0	active
slack variables	w	w	0	$+\infty$	`lower`	0	active

(b)

Figure 7.2: (a) An example input to the CLP(Q) solver. (b) Solved form computed by CLP(Q) for the input. x_4 and x_2 are variables of first kind. x_3 is variable of second kind. x_1, u, w, and v are variables of third kind.

Global data structures The global maps X, *Def*, *Low*, *Up*, *Active*, and *Val* are components of the solved form. They are initialized to be empty. So initially the solved form is empty. Note that when we pick a fresh variable x_k at line 11 of AddInequality that means x_k is not referred by any of the input constraints, and x_k is not in current X. During the run of CLP(Q), some variables are detected to be equal to a constant. Such equalities are stored in *queue* and these equalities are added to the solved form at the end of execution of the entry procedures (In AddInequality, lines 13–15 and in AddInequality lines 21–23).

Now we will describe procedures of the algorithm.

Procedures Deref and Initialize Both the entry procedures call Deref to de-reference the term of the input atom. Deref replaces each variable appearing in the input term with its definition in the solved form. If a variable in not yet part of the solved form then the procedure Initialize is called to add the variable in the solved form as a non-basis, undefined, inactive, and unbounded variable. Deref eliminates all defined variables from the input term and returns a term over undefined variables.

Procedure Substitute This procedure takes an undefined variable x_m and a term over other undefined variables as input. Substitute replaces each occurrence of x_m in the definitions by the given input term. These replacements may leads to violation of condition (8). So Substitute also updates *Val* such that condition (8) holds at the end of this procedure. As a result, Substitute turns x_m into a defined variable.

Procedure Pivot The inputs of this procedure are a basis variable x_b, an activation direction *act*, and an undefined and active variable x_i that appears in the definition of x_b. Pivot removes x_b from the basis and adds x_i to the basis using Substitute at lines 1–3. x_b is now an undefined variable that appears in the definition of the basis variable x_i, so x_b has to be made active. Pivot activates x_b in the direction *act* by calling procedure Activate at line 5. Since x_i is added to the basis, x_i is made inactive at line 6.

Procedures Activate and AddBasis Activate activates an inactive variable. It also has to update *Val* to satisfy condition (7) and (8). The inputs of AddBasis are a defined variable x_m and an activation direction *act*. This procedure add x_m to basis and makes it inactive. Each variable x_j appearing in the definition of x_m is activated at lines 4–13. If either of the bounds of x_j does not exist then the other bound is activated at lines 6–9. Otherwise, if c_j is positive then x_j is activated in the direction *act* and if c_j is negative then x_j is activated in the direction opposite to *act* at lines 10–13. $\oplus$ denotes the logical xor operator.

Procedure AddEquality This procedure takes a linear equality $t = 0$ as input. At line 1, t is de-referenced using the solved form. If the solved form implies $t = 0$ then the condition at line 2 is true and procedure continues at line 13. If the condition at line 4 is true then the conjunction of the solved form and

```
global variables
  { X = ∅        : set of variables                  Basis = ∅  : set of variables   }
  { Def = ∅      : X → linear terms                                                  }
  { Low = ∅      : X → ℚ ∪ {−∞}                      Up = ∅     : X → ℚ ∪ {+∞}       } : solved form
  { Active = ∅   : X → {none, lower, upper}          Val = ∅    : X → ℚ              }
  queue = ∅: set of linear atoms
```

```
procedure AddEquality
input
  $t = 0$ : linear constraint
begin
  $c + \sum_{j\in J} c_j x_j$ := Deref($t$)
  if $c = 0 \wedge J = \emptyset$ then
      skip
  elsif $c \neq 0 \wedge J = \emptyset$ then
      throw "Unsatisfiable"
  elsif $\exists i \in J.\ Low(x_i) = -\infty \wedge Up(x_i) = +\infty$ then
      Substitute($x_i, -\frac{1}{c_i}(c + \sum_{j\in J\setminus\{i\}} c_j x_j)$)
  else
      pick $i \in J$
      Substitute($x_i, -\frac{1}{c_i}(c + \sum_{j\in J\setminus\{i\}} c_j x_j)$)
      AddBasis($x_i$, lower)
      RepairBasis()
  if $s = 0 \in queue$ then
      $queue := queue \setminus \{s = 0\}$
      AddEquality($s = 0$)
end
```

```
procedure AddInequality
input
  $t \leq 0$ : linear constraint
begin
  $c + \sum_{j\in J} c_j x_j$ := Deref($t$)
  if $c \leq 0 \wedge J = \emptyset$ then
      skip
  elsif $c > 0 \wedge J = \emptyset$ then
      throw "Unsatisfiable"
  elsif $t = a + a_i x_i$ then
      UpdateBound($a + a_i x_i \leq 0$)
  elsif $J = \{j\}$ then
      UpdateBound($c + c_j x_j \leq 0$)
  else
      pick fresh $x_k$                    (* slack variable *)
      Initialize($x_k$)
      $Low(x_k)$ := 0
      if $\exists i \in J.Low(x_i) = -\infty \wedge Up(x_i) = +\infty$ then
        Substitute($x_i, -\frac{1}{c_i}(c+\sum_{j\in J\setminus\{i\}} c_j x_j + x_k)$)
      else
        $Def(x_k)$ := $-(c + \sum_{j\in J} c_j x_j)$
        $Val(x_k)$ := $-(c + \sum_{j\in J} c_j\, Val(x_j))$
        AddBasis($x_k$, lower)
        RepairVar($x_k$)
  if $s = 0 \in queue$ then
      $queue := queue \setminus \{s = 0\}$
      AddEquality($s = 0$)
end
```

```
procedure Initialize
input
  $x_i$ : uninitialized variable
begin
  $X$ := $X \cup \{x_i\}$
  $(Def(x_i), Active(x_i), Val(x_i))$ := $(x_i$, none, $0)$
  $(Low(x_i), Up(x_i))$ := $(-\infty, +\infty)$
end
```

```
procedure Deref
input
  $c+\sum_{j\in J} c_j x_j$ : linear term
begin
  $t$ := $c$
  for each $j \in J$ do
      if $Def(x_j) = \bot$ then
        Initialize($x_j$)
      $t$ := $t + c_j Def(x_j)$
  return $t$
end
```

```
procedure Substitute
input
  $x_m$ : undefined variable
  $c + \sum_{j\in J} c_j x_j$ : linear term
begin
  $d$ := $c + \sum_{j\in J} c_j\, Val(x_j) - Val(x_m)$
  for each $x_k \in X$ :
  $a + \sum_{i\in I} a_i x_i = Def(x_k) \wedge m \in I$ do
      $Val(x_k)$ := $Val(x_k) + a_m d$
      $Def(x_k)$ := $a+ \sum_{i\in I\setminus\{m\}} a_i x_i +$
                          $a_m(c + \sum_{j\in J} c_j x_j)$
end
```

```
procedure Pivot
input
  $x_b$ : basis variable
  $act$ : activation direction
  $x_i$ : undefined and active variable
begin
  $c + \sum_{j\in J} c_j x_j$ := $Def(x_b)$
  $t$ := $-\frac{1}{c_i}(c + \sum_{j\in J\setminus\{i\}} c_j x_j - x_b)$
  Substitute($x_i, t$)
  $Basis$ := $(Basis \setminus \{x_b\}) \cup \{x_i\}$
  Activate($x_b, act$)
  $Active(x_i)$ := none
end
```

Figure 7.3: Algorithm in CLP(Q) page 1

```
procedure UpdateBound
input
  $c + c_j x_j \leq 0$ : single variable linear inequality
begin
  if $c_j > 0$ then
     $Status$ := UpdateUpper($x_j, -c/c_j$)
     $act$ := lower
  else
     $Status$ := UpdateLower($x_j, -c/c_j$)
     $act$ := upper
  if $Status =$ updated $\wedge$ $Def(x_j) \neq x_j$ then
     if $x_j \notin Basis$ then
       $a + \sum_{i \in I} a_i x_i$ := $Def(x_j)$
       if $\exists k \in I$ $Low(x_k) = -\infty \wedge Up(x_k) = +\infty$ then
          Substitute($x_k, -\frac{1}{a_k}(a + \sum_{i \in J \setminus \{k\}} a_i x_i - x_j)$)
       else
          AddBasis($x_j, act$)
     if $x_j \in Basis$ then
       RepairVar($x_j$)
end
```

```
procedure UpdateLower
input
  $x_j$ : variable
  $lb$ : $\mathbb{Q}$
begin
  if $Up(x_j) < lb$ then
    throw "Unsatisfiable"
  elsif $Up(x_j) = lb$ then
    EnQueue( $x_j = lb$ )
  else $Low(x_j) < lb$ then
    if $Active(x_j) =$ lower then
      _ := PushUp($x_j$)
    $Low(x_j)$ := $lb$
    if $Active(x_j) =$ lower then
      Activate($x_j$, lower)
    return updated
  return noChange
end
```

```
procedure UpdateUpper
input
  $x_j$ : variable
  $ub$ : $\mathbb{Q}$
begin
  if $Low(x_j) > ub$ then
    throw "Unsatisfiable"
  elsif $Low(x_j) = ub$ then
    EnQueue( $x_j = ub$ )
  else $Up(x_j) > ub$ then
    if $Active(x_j) =$ upper then
      _ := PushLow($x_j$)
    $Up(x_j)$ := $ub$
    if $Active(x_j) =$ upper then
      Activate($x_j$, upper)
    return updated
  return noChange
end
```

```
procedure AddBasis
input
  $x_m$ : variable entering in basis
  $act$ : preferred activation
begin
  $Basis$ := $Basis \cup \{x_m\}$
  $Active(x_m)$ := none
  $c + \sum_{j \in J} c_j x_j$ := $Def(x_m)$
  for each $j \in J$ : $Active(x_j) =$ none
  do
    if $Low(x_j) = -\infty$ then
        Activate($x_j$, upper)
    elsif $Up(x_j) = +\infty$ then
        Activate($x_j$, lower)
    elsif $act =$ upper $\oplus$ $c_j > 0$ then
        Activate($x_j$, lower)
    else
        Activate($x_j$, upper)
end
```

```
procedure Activate
input
  $x_m$ : variable
  $act$ : activation direction
begin
  $Active(x_m)$ := $act$
  match $act$ with
   | lower -> $new$ := $Low(x_m)$
   | upper -> $new$ := $Up(x_m)$
  $d$ := $new - Val(x_m)$
  for each $x_k \in X$ :
  $c + \sum_{i \in I} c_i x_i = Def(x_k) \wedge m \in I$ do
    $Val(x_k)$ := $Val(x_k) + c_m d$
end
```

```
procedure EnQueue
input
  $x_k = c$ : variable equality
begin
  $queue$ := $queue \cup \{x_k - c = 0\}$
end
```

Figure 7.4: Algorithm in CLP(Q) page 2

```
procedure RepairBasis
begin
  $LocalBasis := Basis$
  for each $x_b \in LocalBasis \wedge x_b \in Basis$ do
    RepairVar($x_b$)
end
```

```
procedure RepairUp
input
  $x_b$ : basis variable
begin
  if $Val(x_b) < Up(x_b)$ then   return
  $c + \sum_{i \in I} c_i x_i := Def(x_b)$
  if $\exists i \in I.\ c_i > 0 \wedge Active(x_i) =$ upper then
    $Status$ := PushLow($x_i$)
  elsif $\exists i \in I.\ c_i < 0 \wedge Active(x_i) =$ lower then
    $Status$ := PushUp($x_i$)
  else
    $Status$ := optimum
    if $Val(x_b) = Up(x_b)$ then
      EnQueue( $x_b = Up(x_b)$ )
    else
      throw "Unsatisfiable"
  match $Status$ with
   | applied -> RepairUp($x_b$)
   | nobound($x_i$) -> Pivot($x_b$, upper, $x_i$)
end
```

```
procedure PushLow
input
  $x_i$ : undefined and active variable
begin
  $(lb, k) := (Low(x_i) - Up(x_i), i)$
  for each $x_b \in Basis$ :        (* Pick $x_b$ in order *)
  $Def(x_b) = c + \sum_{j \in J} c_j x_j \wedge i \in J$ do
    if $c_i > 0 \wedge \frac{Low(x_b) - Val(x_b)}{c_i} > lb$ then
      $(lb, k, act) := (\frac{Low(x_b) - Val(x_b)}{c_i}, b,$ lower$)$
    if $c_i < 0 \wedge \frac{Up(x_b) - Val(x_b)}{c_i} > lb$ then
      $(lb, k, act) := (\frac{Up(x_b) - Val(x_b)}{c_i}, b,$ upper$)$
  done
  if $k = i \wedge Low(x_i) = -\infty$ then
    return nobound($x_i$)
  elsif $k = i$ then
    Activate($x_i$, lower)
  else
    Pivot($x_k, act, x_i$)
  return applied
end
```

```
procedure RepairVar
input
  $x_b$ : basis variable
begin
  if $Val(x_b) \geq Up(x_b)$ then   RepairUp($x_b$)
  if $Val(x_b) \leq Low(x_b)$ then   RepairLow($x_b$)
end
```

```
procedure RepairLow
input
  $x_b$ : basis variable
begin
  if $Val(x_b) > Low(x_b)$ then   return
  $c + \sum_{i \in I} c_i x_i := Def(x_b)$
  if $\exists i \in I.\ c_i > 0 \wedge Active(x_i) =$ lower then
    $Status$ := PushUp($x_i$)
  elsif $\exists i \in I.\ c_i < 0 \wedge Active(x_i) =$ upper then
    $Status$ := PushLow($x_i$)
  else
    $Status$ := optimum
    if $Val(x_b) = Low(x_b)$ then
      EnQueue( $x_b = Low(x_b)$ )
    else
      throw "Unsatisfiable"
  match $Status$ with
   | applied -> RepairLow($x_b$)
   | nobound($x_i$) -> Pivot($x_b$, lower, $x_i$)
end
```

```
procedure PushUp
input
  $x_i$ : undefined and active variable
begin
  $(ub, k) := (Up(x_i) - Low(x_i), i)$
  for each $x_b \in Basis$ :        (* Pick $x_b$ in order *)
  $Def(x_b) = c + \sum_{j \in J} c_j x_j \wedge i \in J$ do
    if $c_i > 0 \wedge \frac{Up(x_b) - Val(x_b)}{c_i} < ub$ then
      $(ub, k, act) := (\frac{Up(x_b) - Val(x_b)}{c_i}, b,$ upper$)$
    if $c_i < 0 \wedge \frac{Low(x_b) - Val(x_b)}{c_i} < ub$ then
      $(ub, k, act) := (\frac{Low(x_b) - Val(x_b)}{c_i}, b,$ lower$)$
  done
  if $k = i \wedge Up(x_i) = +\infty$ then
    return nobound($x_i$)
  elsif $k = i$ then
    Activate($x_i$, upper)
  else
    Pivot($x_k, act, x_i$)
  return applied
end
```

Figure 7.5: Algorithm in CLP(Q) page 3

$t = 0$ is unsatisfiable, and we throw an exception "`Unsatisfiable`". If the de-referenced term contains an undefined and unbounded variable then this variable is substituted by the rest of de-referenced term in the solved form at line 7 and we get a solved form. Otherwise, we pick a variable appearing in the de-referenced term and the variable is substituted by the rest of de-referenced term in the solved form. Since, the variable is bounded and defined, we add the variable in basis at line 11. Note that we pass `lower` as the activation direction in the second argument of AddBasis, which is an arbitrary choice. These modifications of the solved form may leads to violation of condition (9), which is fixed by calling RepairBasis. The code after line 13 is already discussed in description of the global data structures.

Procedure AddInequality This procedure takes a linear inequality $t \leq 0$ as input. At line 1, t is de-referenced using the solved form. If the solved form implies $t \leq 0$ then the condition at line 2 is true and procedure continues at line 21. If the condition at line 4 is true then the conjunction of the solved form and $t \leq 0$ is unsatisfiable, and we throw an exception "`Unsatisfiable`". If either the input term or de-referenced term contains a single variable then UpdateBound is called at line 7 or 9, respectively. Otherwise, we introduce a slack variable x_k and initialize the lower bound of x_k with 0 at lines 11–13. Now we need to add an equality between x_k and the negation of the de-referenced term in the solved form. If the de-referenced term contains an undefined and unbounded variable then this variable is substituted by the rest of de-referenced term added with x_k in the solved form at line 15 and we get a solved form. Otherwise, AddInequality sets definition of x_k to negation of the de-referenced term and add x_k to the basis at line 17 and 19. Only x_k can violate condition (9). So RepairVar is called to fix the violation at line 20. The code after line 21 is already discussed in description of the global data structures.

Procedures RepairBasis and RepairVar RepairBasis iteratively changes the basis by pivot operations until condition (9) is satisfied. RepairBasis keeps a local copy of the current basis and then iterate over the variables that will remain in the basis after the call to RepairVar in each iteration. RepairVar takes a basis variable x_b as input, checks if $\mathit{Val}(x_b)$ violates any of its bounds, and calls accordingly RepairUp or RepairLow, accordingly.

Procedures RepairUp, RepairLow, PushLow, and PushUp We only discuss RepairUp and PushLow. The descriptions of RepairLow and PushUp are similar, respectively.

RepairUp takes a basis variable x_b as input. RepairUp recursively attempts to decrease $\mathit{Val}(x_b)$ such that $\mathit{Val}(x_b) < \mathit{Up}(x_b)$ or moves x_b out of the basis. The condition (8) defines $\mathit{Val}(x_b)$ in terms of values of variables appearing in $\mathit{Def}(x_b)$. The condition at line 3 holds if a variable x_i appears in $\mathit{Def}(x_b)$ with a positive coefficient and is activated with the direction `upper`. We can decrease $\mathit{Val}(x_b)$ by decreasing $\mathit{Val}(x_i)$. Since $\mathit{Val}(x_i)$ is taking the maximum allowed value, it can be decreased. At line 4, procedure PushLow is called to decrease value of $\mathit{Val}(x_i)$. The code at line 5 and 6 is symmetric therefore we will not discuss it. If both conditions at line 3 and 5 fail then we can not decrease $\mathit{Val}(x_i)$ any further, and the execution continues at line 8. If $\mathit{Val}(x_b)$ is equal to $\mathit{Up}(x_b)$ then we have detected an equality and this equality is pushed into *queue*. Otherwise, the solved form is unsatisfiable and exception "`Unsatisfiable`" is thrown. The return value of the call to PushLow at line 4 is stored in *Status*. *Status* equals to `applied` indicates that a progress in decreasing $\mathit{Val}(x_b)$ has been made. Then, we decrease $\mathit{Val}(x_b)$ further. *Status* equals to `nobound`(x_i) indicates that x_i can be decreased without any bound and by doing a pivot operation between x_b and x_i we can satisfy the conditions of the solved form.

In procedure PushLow at line 1, k is set equal to i and lb records the maximum change in value of $\mathit{Val}(x_i)$ allowed by $\mathit{Low}(x_i)$. Then at lines 2–8, PushLow iterates over the basis variables and finds a basis variable x_k that may impose maximum bound on smallest value of lb, i.e., change in value of $\mathit{Val}(x_i)$. There are three possible cases at lines 9–14. The first and second case occur when no bounding basis variable exists and k remains equal to i at line 9. The first case occurs if there is no lower bound of x_i. In this case, a value nobound(x_i) is returned indicating that $\mathit{Val}(x_i)$ can be decreased without any bound at line 10. The second case occurs if there is a lower bound on x_i. In this case, the activation direction of x_i is changed from `upper` to `lower` at line 12. This change leads to a decrease of $\mathit{Val}(x_b)$. The third case occurs if x_k is a basis variable. x_k is leaves the basis and x_i enters the basis at line 14. After the second and third cases, the execution continues at line 15 where `applied` is returned indicating to the caller that $Val(x_b)$ is decreased by some amount.

Procedures UPDATEBOUND, UPDATELOWER, and UPDATEUPPER The procedure UPDATEBOUND takes an inequality that contains only one variable as input and updates bounds of this variable. Depending on the variable coefficient in the input inequality, upper or lower bound is updated by calling UPDATEUPPER or UPDATELOWER, respectively.

We will discuss UPDATEUPPER. The description of UPDATELOWER is symmetric. UPDATEUPPER takes a variable x_j and a new upper bound *ub* for x_j as input. If *ub* is strictly lower than the lower bound of x_j then UPDATEUPPER throws an exception "`Unsatisfiable`" at line 2. If *ub* is equal to the lower bound of x_j then we have detected that x_j to be constant and the corresponding constant equality is stored in *queue* at line 4. If $Up(x_j) > ub > Low(x_j)$ then we update $Up(x_j)$. Due to conditions (7)–(9), updating an active bound is a difficult case. If $Active(x_j) = \texttt{upper}$ then PUSHLOW is called at line 7. If PUSHLOW moves x_j into the basis or changes its activation direction then the difficulty is eliminated. Otherwise, PUSHLOW makes no changes in solved form and solved form imposes no limit in decrease of upper bound. In both case, we update $Up(x_j)$ at line 8 without violating conditions (7)–(9) for any other variable. If $Active(x_j)$ is still equal to `upper` at line 9 then we update *Val* by calling ACTIVATE to satisfy condition (9). UPDATEUPPER returns value `updated` only upper bound is changed otherwise `noChange` is returned to the caller, i.e., UPDATEBOUND.

In UPDATEBOUND at line 7, if a bound of x_j is updated and x_j is a defined variable then lines 8–13 are executed to maintain condition (2) and (3). At line 14, if x_j is in the basis then we check and repair any violation of condition (9).

CLP(LI+UIF) using CLP(Q)

Figure 7.6 presents the CLP(LI+UIF) as an extension of CLP(Q). CLP(LI+UIF) solver extends CLP(Q) solver with a congruence checker for uninterpreted functions. The CLP(LI+UIF) contains an additional data structure *TermDef* that is a function from pairs of uninterpreted function symbols and lists of linear terms to a variable. *TermDef* is used to purify input atoms to produce linear atoms, and to check if a congruence axiom can be applied on input constraints and to produce new equalities. CLP(LI+UIF) takes $\mathcal{T}_{\mathrm{LI+UIF}}$ atoms as the input sequence. Given an input, the current solved form, the current *TermDef*, CLP(LI+UIF) computes the next solved form and *TermDef*. If CLP(LI+UIF) fails to compute the next solved form and *TermDef* then it throws an exception "`Unsatisfiable`". CLP(LI+UIF) adds the following three procedures.

Procedure ADDCONSTRAINT At line 1, PURIFY is called to remove uninterpreted functions from the input term and to produce a linear term. Next, the purified atom is added to CLP(Q) solver using its entry procedures at lines 2–4. If the call to an entry procedure of CLP(Q) does not throw an exception "`Unsatisfiable`" then CONGCHK is called at line 5 to check if congruence rules can be applied between any two of the terms stored in *TermDef*.

Procedure PURIFY This procedure takes a term in $\mathcal{T}_{\mathrm{LI+UIF}}$. PURIFY recursively traverses the input term in the bottom up order. During the traversal, PURIFY replaces each subterm whose top function symbol is uninterpreted with a variable. If the subterm is already seen before then the variable corresponding to the subterm is retrieved from *TermDef* at line 12. Otherwise, a fresh variable is chosen to replace for the subterm, and *TermDef* is updated accordingly at lines 9 and 10.

Procedure CONGCHK This procedure recursively executes until no new equality is detected from the solved form and *TermDef*. At lines 1–5, the new equalities are detected by the following if-condition. Let two variables x_j and x_k be in the range of *TermDef*. Assuming that in *TermDef*, x_j and x_k are mapped by the same function symbol f and lists of subterms $s_1, \ldots, s_m$ and $t_1, \ldots, t_m$, respectively. For all $i \in 1..m$, if the solved form implies $s_i = t_i$, which is checked by call to DEREF, then due to the congruence rule, $x_j = x_k$.