CLAM Specification for Provably Correct Compilation of CLP($\mathcal{R}$) Programs

Egon Börger and Rosario F. Salamone

Abstract

The chapter extends the correctness proof in [4] for compilation of Prolog programs on the WAM to CLP($\mathcal{R}$) programs on the Constraint Logic Arithmetic Machine (CLAM [8, 10]). This serves to illustrate, through a complex case study, how the evolving algebra specification methodology allows us to incorporate modularity and extendability principles in system design.

1 Introduction

This chapter extends, to the Constraint Logic Arithmetic Machine (CLAM) and CLP($\mathcal{R}$) programs, the mathematical analysis of the Warren Abstract Machine (WAM) for executing Prolog and the resulting correctness proof for a general compilation scheme of Prolog to the WAM given in [4]. Starting from an abstract CLP($\mathcal{R}$) model—which paraphrases the primary model for Prolog defined in [5]—we follow the stepwise refinement of Prolog models to the WAM model given in [4] and enrich both the specification and the correctness proofs by what is needed to cover CLP($\mathcal{R}$) constraints and their implementation in the CLAM. Use of Gurevich's evolving algebras [6] as our specification vehicle allows us to couple smoothly the modularity of the WAM specification in [4] to extendibility of unification to handling of CLP($\mathcal{R}$) constraints.

For expository reasons we start (in section 3) from scratch by defining an operational semantics of CLP($\mathcal{R}$) through *CLP($\mathcal{R}$) trees*, close to the usual intuitive picture and to its proof theoretical logical background (resolution-trees). This definition, which serves as our primary mathematical model, is easily shown to be correct w.r.t. resolution (for CLP($\mathcal{R}$) programs without built-in predicates), and can be extended to a transparent rigorous formulation of the full language using the definitions given in [5].

The first major refinement steps are devoted to analysis and implementation of the disjunctive and conjunctive structure of CLP($\mathcal{R}$) programs. These structures correspond to WAM handling of clause selection (predicate structure) and continuations (clause structure) and can be taken unchanged from the WAM as long as the representation of terms and constraints is kept abstract. We then analyze in detail how the WAM representation of terms and substitutions has to be enriched in order to make it work also for constraints (sections 4), the cooperation of unification and contraint solving (section 5) and constraint compilation (section 6). The fine points of the WAM—environment trimming, local and unsafe values, last call optimization, Warren's classification of variables and their on-the-fly initialization—can again be taken almost unchanged from the WAM and enriched by specific CLAM optimizations (section 7).

We thus arrive at a specification of the full CLAM which allows us to prove the following:

Theorem 1.1. (Main Theorem) *Each compiler which satisfies the assumptions for predicate, clause, term compilation listed in [4] and the assumptions for constraint compilation listed in this chapter, compiles CLP($\mathcal{R}$) programs correctly with respect to CLP($\mathcal{R}$) trees and CLAM algebras.*

All we assume of the reader is general understanding of CLP($\mathcal{R}$) and Prolog as programming languages. In order to keep the chapter within reasonable bounds we refer to [4] each time we can take from there without crucial modifications. To help readability we list, however, in section 2 the basic notions from evolving algebras.

2 Prerequisites on Evolving Algebras

The CLAM model constructed in this chapter is an *evolving algebra*, a notion introduced by Gurevich in [7]. Since this notion is a mathematically rigorous form of fundamental operational intuitions of computing, the chapter can be followed without any particular theoretical prerequisites: indeed our rules can be read as "pseudocode over abstract data". For completeness we nevertheless list in this section the basic definitions for evolving algebras.

The abstract data come as elements of sets (domains, *universes*), and the allowed operations as partial *functions*. This determines a class of algebras (in the sense used in logic) or states of an abstract "machine" which we allow to *evolve* in time by executing *function updates* of form $f(t_1, \ldots, t_n) := t$ whose execution is to be understood as *changing* (or defining, if there was none) the value of function f at given arguments.

Definition 2.1. *An* evolving algebra *is a finite set of* transition rules *of form*

$$\textbf{if } R? \textbf{ then } R!$$

where $R?$ (condition or guard) is a boolean, the truth of which triggers simultaneous *execution of all updates in the finite set $R!$ of function updates.*

Simultaneous execution avoids fussing with intermediate storage problems. Since functions may be partial, equality in the guards is to be interpreted as implying that both arguments are defined. The signature of an evolving algebra can always be reconstructed, as the set of function symbols occurring in the rules. An evolving algebra usually comes together with a set of *integrity constraints*, i.e. extralogical axioms and/or rules of inference, specifying the intended domains.

In applications of evolving algebras as here, one usually encounters a *heterogenous* signature with several universes, which may in general grow and shrink in time—update forms are provided to extend a universe:

extend A **by** $t_1, \ldots, t_n$ **with** *updates* **endextend**

where *updates* may (and should) depend on t_i's, setting the values of some functions on *newly created* elements t_i of A. In [6] Gurevich has shown how to reduce such setups to the above basic model of a homogenous signature (with one universe) and function updates only.

The forms obviously reducible to the above basic syntax, which we shall freely use as abbreviations, are **let**, **case** and **if then else**. We shall assume that we have the standard mathematical universes of booleans, integers, lists of whatever etc. (as well as the standard operations on them) at our disposal without further mention. We use usual notations, in particular Prolog notation for lists.

An evolving algebra, as given above, determines the dynamics of a very large transition system. Here we are usually interested only in states reachable from some designated *initial states*, which may be specified in various ways. We use an informal mathematical description, like in model theory; but one could easily devise special initializing evolving algebra rules which, starting ¿from a canonical "empty" state, produce the initial states we need.

As CLP($\mathcal{R}$) is a sequential language, our rules are organized in such a way that at every moment at most one rule is applicable.

3 CLP($\mathcal{R}$) with Predicate and Clause Compilation

In this section we show how one can adapt the Prolog tree model of [5] to CLP($\mathcal{R}$) ([8]) and its refinement with predicate and clause compilation. We suppose the reader to be acquainted with the fundamentals of Prolog and of CLP($\mathcal{R}$).

To obtain a primary model for CLP($\mathcal{R}$), characterized by the programmer's view and close to resolution trees, it suffices to paraphrase the Prolog model of [5] by replacing substitution by constraints and unification by constraint solving, similar to what has been done for the type constraints extension of Prolog in [2].

In the following we define in some detail the primary CLP($\mathcal{R}$) model and then explain shortly how to refine this model to the level where predicates and clauses are no longer abstract but compiled. The reader who is acquainted with [5] or [2] might skip this section.

3.1 CLP Trees

A CLP($\mathcal{R}$) computation can be seen as systematic search of a space of possible solutions to an initially given query. The set of computation states is viewed as carrying a tree structure, with initial state at the root, and *son* relation representing alternative (single) derivation steps. This means to represent *CLP($\mathcal{R}$) computation states* in a set *NODE* with two distinguished elements *root* and *currnode*, with the latter representing the (dynamically) current state. Each element of *NODE* has to carry all information relevant—at the desired abstraction level—for the computation state it represents. This information consists of *the sequence of goals* still to be executed, the *set of constraints* collected so far, and possibly the *sequence of alternative derivation states* still to be tried, as explained below.

The tree structure over the universe *NODE*, representing the structure of CLP($\mathcal{R}$)'s *backtracking behavior*, is realized by a total function

$$father \quad : \quad NODE - \{root\} \;\rightarrow\; NODE$$

such that from each node there is a unique *father* path towards *root*.

When, at a given node n, a user defined atom is selected (as activator *act*) for execution, for each possible immediate derivation state a son of n will be created, to control the alternative computation thread. Each son is determined by a corresponding *candidate clause* of the program, i.e. one of those clauses whose head might unify with *act*. All such *candidate sons* are attached to n as a list $cands(n)$, in the order reflecting the ordering of corresponding candidate clauses in the program. We require of course the *cands* lists to be consistent with *father*, i.e. whenever *Son* is among $cands(Father)$, then $father(Son) = Father$.

This action of augmenting the tree with $cands(n)$ takes place at most once, when n gets first visited (in *Call mode*). The mode then turns to *Select*, and the first unifying son from $cands(n)$ gets visited (i.e. becomes the value of *currnode*), again in *Call* mode. The selected son is simultaneously deleted ¿from the $cands(n)$ list. If control ever returns to n (by *backtracking*, cf. below), it will be in *Select* mode, and the next candidate son will be selected, if any.

If none, that is if in *Select* mode $cands(n) = [\,]$, all attempts at derivation from the state represented by n will have failed, and n will be *abandoned* by returning control to its *father*. This action is usually called *backtracking*.

The information relevant for determining a computation state will be associated to nodes by appropriate (in general partial) functions on the universe *NODE*. For each state we have to know the sequence of goals still to be executed. In view of the *cut* operator *!*, however, this sequence is not represented linearly, but structured into subsequences—clause bodies decorated with appropriate *cutpoints*, i.e. backtracking states current when the clause was called: a function $decglseq : NODE \rightarrow DECGOAL^*$ for $DECGOAL = (LIT + CONSTRAINT)^* \times NODE$ associates the relevant sequence of (decorated) goals to each node. CONSTRAINT is the universe of constraints coming with functions

$$c \ : \ NODE \rightarrow CS, \quad solvable \ : \ CS \rightarrow BOOL$$

where CS is the universe of all sets of constraints.

The above-mentioned switching of *modes* will be represented by a distinguished element $mode \in \{Call, Select\}$ indicating the action to be taken at *currnode*: creating the derivation states, or selecting among them. To be able to speak about *termination* we will use a distinguished element $stop \in \{0, 1, -1\}$, to indicate respectively running of the system, halting with success and final failure. We will use (and consider as part of CLP tree algebras) all the usual CLP($\mathcal{R}$) data structures and *list operations* for which we adopt standard notation. In the same way we shall use *hd* and *bdy* to select heads and bodies of clauses, allowing ourselves the freedom to confuse a list of literals and constraints with their iterated conjunction. The codomain of *bdy* will thus be taken to be $(TERM \bigcup CONSTRAINT)^*$.

We keep the above-mentioned notion of *candidate clause* (for executing an atom) abstract (regarding it as implementation defined), assuming only the following integrity constraints: every candidate clause for a given atom has the proper predicate (symbol), i.e. the same predicate as the atom (*correctness*); every clause whose head unifies with the given atom is candidate clause for this atom (*completeness*). The reader might think of considering any clause occurrence whose head is formed with the given predicate, or the clause occurrences selected by an indexing scheme, or just all occurrences of unifying clauses.

Having to allow for *dynamic code* and related operations, one has to speak about different occurrences of clauses in a program. We hence introduce an abstract universe $CODE$ of clause occurrences (or pointers), coming with functions

$$clause \ : \ CODE \rightarrow CLAUSE, \quad cll \ : \ NODE \rightarrow CODE$$

where $cll(n)$ is the candidate clause occurrence ("clauseline") corresponding to a candidate son n of a computation state, and $clause(p)$ is the clause "pointed at" by p. Note that we do not assume any ordering on $CODE$. We instead assume an abstract function

$$procdef \ : \ LIT \times PROGRAM \times CONSTRAINT \rightarrow CODE^*,$$

which we assume to yield the (properly ordered) list of the candidate clause occurrences for the given literal in the given program (the constraint parameter will allow us to formalize indexing mechanisms which depend on constraints). The current program is represented by a distinguished element *db* of *PROGRAM* (the *database*). Note that existence of *procdef* is all that we assume of the abstract universe *PROGRAM*.

We assume the following **initialization** for application of the rules given below:

- *root* is supposed to be the nil element—on which no function is defined—and father of *currnode*;
- *currnode* has a one element list $[\langle \mathit{query}, \mathit{root} \rangle]$ as decorated goal sequence, and empty set of constraints;
- the mode is *Call*, *stop* has value 0; *db* has the given program as value;
- the *cands* list is not (yet) defined at *currnode*.

We now define the five basic rules by which the system attempts to reach a state with $\mathit{stop} = 1$ (due to first successful execution of the query) or with $\mathit{stop} = -1$ (due to its final failure by backtracking all the way to *root*). In writing these rules, we suppress the parameter *currnode* by simply writing *father* for $\mathit{father}(\mathit{currnode})$, *cands* for $\mathit{cands}(\mathit{currnode})$, *c* for $c(\mathit{currnode})$ and *decglseq* for $\mathit{decglseq}(\mathit{currnode})$. Components of decorated goal sequence are accessed as $\mathit{goal} \equiv \mathit{fst}(\mathit{fst}(\mathit{decglseq}))$, $\mathit{cutpt} \equiv \mathit{snd}(\mathit{fst}(\mathit{decglseq}))$, $\mathit{act} \equiv \mathit{fst}(\mathit{goal})$[1], $\mathit{cont} \equiv [\langle \mathit{rest}(\mathit{goal}), \mathit{cutpt} \rangle \mid \mathit{rest}(\mathit{decglseq})]$, with *act* standing for the selected literal (*activator*), and *cont* for *continuation.* We also use the following abbreviation:

```
backtrack  ≡  if $\mathit{father} = \mathit{root}$ then $\mathit{stop} := -1$
              else $\mathit{currnode} := \mathit{father}$
                   $\mathit{mode} := \mathit{Select}$
```

When the *stop* value is 1 or -1, no transition rule will be applicable which is a natural notion of "terminating state". All transition rules will thus be tacitly assumed to stand under the guard

$$\text{OK} \equiv \mathit{stop} = 0$$

The following **query success rule**—for successful halt— leads to successful termination when all goals have been executed:

$$\textbf{if } \mathit{all\_done} \textbf{ then } \mathit{stop} := 1$$

where *all_done* abbreviates $\mathit{decglseq} = [\,]$ [2]. The following **goal success rule** describes success of a clause body, when the system continues to

[1]This definition implements the left-to-right computation rule. In general we could have written $\mathit{act} = \mathit{goal\_select}(\mathit{goal})$, where *goal_select* represents the computation rule. Correspondingly, one has to change the definition of *rest* in *cond.*

[2]Note that we do not describe how output (answer constraints) is given. If the reader wants to view CLP($\mathcal{R}$) as returning all solutions, all he has to do is to modify this rule so as to trigger backtracking.

execute the rest of its goal sequence:

```
if $goal = [\,]$ then $decglseq := rest(decglseq)$
```

The existence of *goal*, assumed in the guard, is understood as excluding *all_done*, cf. above abbreviations. Likewise, the existence of *act*, assumed in rules to follow, is understood as excluding both *all_done* and *goal* = [ ] conditions—we shall, in general, tacitly understand guards, relying on existence of some objects, as excluding all conditions under which these objects could be undefined, suppressing the boolean conditions which would formally ensure such exclusion. In *goal success* rule e.g. the suppressed condition is $NOT(all\_done)$, and in rules mentioning *act* below it is $NOT(all\_done) \,\&\, goal \neq [\,]$.

The crucial derivation step, applicable to a user-defined predicate, is split into *calling* the activator, creating new candidate nodes for alternative derivations from *currnode*, to be followed by *selecting* one of them. We will correspondingly have two rules. The following **call rule**, invoked by having a user-defined activator in *Call* mode, will create as many sons of *currnode* as there are candidate clauses in the procedure definition of its activator, to each of which the corresponding clause(line) will be associated:

```
if   $is\_user\_defined(act)$
   & $mode = Call$
then let $n = length(procdef(act, db))$
       extend $NODE$ by $temp_1, \ldots, temp_n$ with
         $father(temp_i) := currnode$
         $cll(temp_i) := nth(procdef(act, db), i)$
         $cands := [temp_1, \ldots, temp_n]$
       endextend
       $mode := Select$
```

where *is_user_defined* is a boolean function recognizing those literals whose predicate symbols are user defined (as opposed to constraints, built-in predicates and language constructs). Note that goals and constraints are undefined for candidate sons, and that the value of *currnode* does not change[3].

The **Selection Rule** is applicable to nodes already visited in *Call* mode. It triggers the visit of the first candidate node or backtracks if the *cands* list is empty. In the former case, the update of the sequence of goals provides for the unification test and the execution of the body. The currently accumulated constraints are copied and the selected node is cancelled from

[3]Given database operations of full Prolog this rule formalizes the so-called logical view; see [3].

the *cands* list:

```
if $is\_user\_defined(act)$ & $mode = Select$
then if $cands = [\,]$
      then $backtrack$
      else let $clause = rename(clause(cll(fst(cands))), vi)$
           $currnode := fst(cands)$
           $c(fst(cands)) := c$
           $decglseq(fst(cands)) :=$
             $[\langle append([act \doteq hd(clause)], bdy(clause)), father\rangle \mid cont]$
           $cands := rest(cands)$
           $mode := Call$
           $vi := vi + 1$
```

where $p(t_1, \ldots, t_n) \doteq p(s_1, \ldots, s_n)$ abbreviates the sequence $t_1 = s_1, \ldots, t_n = s_n$. Note that we represent renaming of variables abstractly, without going into details of term and variable representation, by introducing a function

$$rename \quad : \quad TERM \times \mathcal{N} \;\rightarrow\; TERM$$

renaming all variables in the given term at the given index (renaming level). The current renaming index—the one to be used for the next renaming—is indicated by a 0-ary function *vi*.

The **Add-constraint Rule** fires when the activator is a constraint. In this case the solvability of the current set of constraints put together with the new constraint is tested; if the answer is yes, then the new constraint *act* is added to *c*; otherwise execution backtracks:

```
if $is\_constraint(act)$ then if $solvable(c \bigcup \{act\})$
                              then $c := c \bigcup \{act\}$
                                   $succeed$
                              else $backtrack$
```

where *succeed* stands for $decglseq := cont$. This concludes the description of the primary CLP($\mathcal{R}$) model as far as user-defined predicates are concerned. For built-in predicates one can now proceed as in Prolog; we refer for this to the full description given in [5] and show here only the example of the cut. It suffices to update *father* to *cutpt*:

```
if $act = !$ then $father := cutpt$
                  $succeed$
```

It is easy to show that this model of CLP($\mathcal{R}$) is correct for CLP($\mathcal{R}$) programs without built-in predicates with respect to the usual resolution based definition of procedural semantics.

3.2 Compilation of Predicates and Clauses

For compilation of predicate and clause structure by WAM instructions, CLP($\mathcal{R}$) constraints behave in the same way as unification equations $s = t$ as long as terms and constraints remain abstract. Therefore the primary (tree based) model for CLP($\mathcal{R}$) of the preceding section can easily be refined to reflect WAM code for creation, reuse and discarding of choicepoints (including switching) and for (de-)allocation of environments (representing clause structure): just apply to the primary CLP($\mathcal{R}$) tree model the same refinement steps as defined (and proved to be correct) for Prolog in [4]. This goes through embedding of the tree into a stack, reuse of choicepoints, determinacy detection, *try- retry- trust-* and *switching*-code (where switching will be canonically extended to arithmetical terms) and environment handling code (where to instructions $call(G)$ also instructions $resolve(C)$ for constraints C will be added). Thus the modularity of the WAM specification in [4] allows us to naturally embed that specification into the CLP($\mathcal{R}$) context and to proceed directly to compilation of terms and constraints.

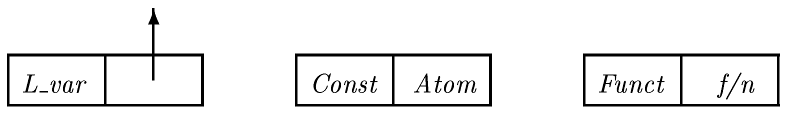
4 Representation of Constraints

When passing to the level of term representation, the CLAM requires to properly extend all corresponding WAM data structures and to introduce new data areas for representation and construction of arithmetic constraints.

CLAM data area locations may contain, in addition to Prolog objects, numeric constants (tagged *Numb*) and "solver variables", i.e. variables appearing in the current collection of arithmetic constraints. Solver variable locations, tagged *S_var*, contain a reference, called the (solver) identifier of that solver variable, to another memory area (*SVAR_AREA*) in which information regarding solver variables becomes accessible. Different solver variables may have the same identifier. Therefore, as in the WAM, we have *DATAAREA*, a set of "locations" with mutually inverse successor and predecessor functions $(+, -)$, and with a "content" function

$$val \quad : \quad DATAAREA \;\rightarrow\; CLPO + MEMORY$$

where, as in the WAM, *MEMORY* is a universe that contains *DATAAREA*, to enable storage of pure pointers, to be elaborated and used below. The set *CLPO* extends the set *PO* of Prolog objects by *SVAR_AREA* + *NUMBER*, coupled with an extension of the WAM encoding scheme



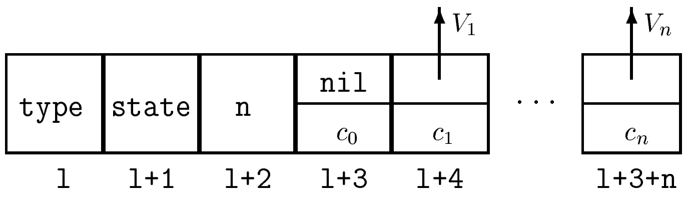


FIG. 1. Inequality $c_0 + c_1V_1 + \ldots + C_nV_n \,\Delta\, 0$, $\Delta \in \{>, \geq\}$.

using functions

$$\begin{array}{lcll} \mathit{tag} & : & \mathit{CLPO} \rightarrow & \{L\_var, S\_var, Numb, Const, List, Struct, Funct\} \\ \mathit{ref} & : & \mathit{CLPO} \rightarrow & DATAAREA + SVAR\_AREA + NUMBER + \\ & & & ATOM + ATOM \times ARITY \end{array}$$

where for uniformity the type label *ref*, used in the WAM to denote logical variables, is replaced by L_var.

For arithmetic constraints we use a stack[4] (*PF_AREA*,*PFO*;*pftop*,*pfbottom*;$+,-$) to store "parametric forms" via a function *pfval* : $PF\_AREA \rightarrow PFO$ as follows.

We call terms of form $c_0 + c_1V_1 + \ldots + c_nV_n$ "linear parametric forms", where c_i is a numeric constant and V_i (called parametric variables) are distinct solver variables with $n \geq 1$. Linear equations or inequalities are of the form $V = lpf$ or $lpf\ \Delta\ 0$, where *lpf* is linear parametric form, $\Delta \in \{\geq, >\}$ and V (called nonparametric variable) is different ¿from the variables in *lpf*. A linear inequality is stored (in consecutive locations starting from l) as shown in Fig. 1, where `type` $\in \{\geq, >\}$; `state` $\in \{active, dormant\}$ indicates whether the linear form refers to a constraint which is actually contained in the current collection of constraints (active linear form), or whether it has been abandoned (in which case it may be restored during backtracking and is called dormant); pointers to the parametric variables V_i are pointers to locations in $SVAR\_AREA$ (i.e. the identifiers of the V_i). Linear equations (`type` is $=$) have one more cell (coming after the arity cell) containing the identifier of the nonparametric variable.

The 0-ary functions $top_=$, $top_\geq$ and $top_>$ represent the (addresses of the) topmost linear equation, nonstrict inequality and strict inequality, respectively. Equations are solved with the Gaussian elimination method. In order to determine the solvability of a set of linear inequalities, an incremental version of the Simplex algorithm is used to maintain inequalities in a solved form representing a certain solution. So there will be another

[4]A stack is used because arithmetic constraints are subject to backtracking.

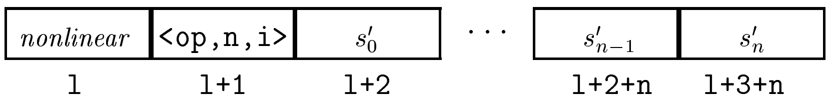


FIG. 2. Nonlinear constraint.

area to store the solved form of each inequality (it is actually a Simplex tableau). In addition, the “inequality solver” also has to determine the equalities implied by a consistent collection of linear inequalities. At this level of abstraction we specify the inequality solver only through abstract updates on which conditions are imposed to ensure correctness.

Nonlinear constraints are stored in nonlinear form, $s_0 = op(s_1, \ldots, s_n)$ where s_j is either a numeric constant or a variable, $0 \leq j \leq n$. A nonlinear constraint stored at location l is represented in Fig. 2, where $\langle op, n, i\rangle \in (ATOM \times ARITY \times \mathcal{N})$ and, if s_j is a constant then $s'_j = s_j$, otherwise s'_j is the identifier of variable s_j. The index i codifies the *wakeup degree.* CLAM delays the satisfiability of nonlinear constraints until they become linear. In order to detect when a nonlinear constraint becomes linear, it is assigned wakeup degrees representing the information currently known about variables appearing in the constraint. As variables become ground, a particular instance of a nonlinear constraint changes degree until it can be awoken.

Nonlinear equations are kept in normal form, in the sense that when a solver identifier gets instantiated to a ground value, it is substituted out in all its “nonlinear” occurrences. The 0-ary function top_{nl} keeps the address of the topmost nonlinear equation.

When a solver variable gets instantiated to an arithmetic term, its value must be substituted in every linear form where it appears and, if the value is ground, also in every nonlinear form. Therefore for each solver variable we need the address of its occurrences and the address of the linear parametric form the variable is possibly bound to. When a solver variable is instantiated to a ground value, this value has to be kept accessible through *SVAR_AREA* because other *S_var* locations in *DATAAREA* may point to the same identifier. Therefore *SVAR_AREA* is realized as a stack (*SVAR_AREA*;*svtop*, *sbottom*; +, -) to store those 3 kinds of objects (accessed using a content function *sval* into *SVARO*):

$$\begin{array}{rcl} svar\_occ & : & SVARO \rightarrow (PF\_AREA \times \mathcal{N})^* \\ svar\_lpf & : & SVARO \rightarrow PF\_AREA + \{nil\} \\ svar\_value & : & SVARO \rightarrow NUMBER + \{nil\} \end{array}$$

Formally we therefore assume that nonparametric variables correspond to identifiers l with $svar\_lpf(sval(l)) \neq nil$ and parametric variables to identifiers with $svar\_lpf(sval(l)) = nil$ & $svar\_value(sval(l)) = nil$ (this corresponds to linear constraints being in solved form). We assume that the elements of the list of occurrences of an identifier are formed by the address of a linear or nonlinear form in which the identifier appears, and its relative position within this form (a sort of offset with respect to the address of the form, for details see [11]).

Notationally, we often suppress the value functions and write $tag(l)$ for $tag(val(l))$, $ref(l)$ for $ref(val(l))$, $svar\_lpf(l)$ for $svar\_lpf(sval(l))$ etc. We borrow from [4] the following abbreviations:

$$l_1 \leftarrow l_2 \equiv val(l_1) := val(l_2),\ l \leftarrow \langle T, R\rangle \equiv tag(l) := T,\ ref(l) := R$$
$$l\_unbound(l) \equiv (\,tag(l) = L\_var \ \&\ ref(l) = l)$$
$$mk\_l\_unbound(l) \equiv l \leftarrow \langle L\_var, l\rangle$$

and define their solver variable analogues by:

$$\begin{array}{rcl} s\_unbound(l) & \equiv & tag(l) = S\_var \ \&\ svar\_lpf(ref(l)) = nil \\ & & svar\_value(ref(l)) = nil \\ s\_unfree(l) & \equiv & tag(l) = S\_var \ \&\ svar\_lpf(ref(l)) \neq nil \\ s\_ground(l) & \equiv & tag(l) = S\_var \ \&\ svar\_value(ref(l)) \neq nil \\ mk\_id\_unbound(l) & \equiv & svar\_lpf(l) := nil, svar\_occ(l) := [], \\ & & svar\_value(l) := nil \\ mk\_s\_unbound(l) & \equiv & l \leftarrow \langle S\_var, svtop\rangle, mk\_id\_unbound(svtop), \\ & & svtop := svtop + \end{array}$$

We extend the WAM dereferencing and term reconstructing functions $deref : DATAAREA \rightarrow DATAAREA$ and $term : DATAAREA \rightarrow TERM$ ¿from [4] to include solver variables. We add the analogues for reconstructing linear parametric forms $lpf : PF\_AREA \rightarrow TERM$ and solver variable values $svar : SVAR\_AREA \rightarrow TERM$. The usual WAM layout for the term represented at location l is thus extended to solver variables and linear parametric forms by the following requirements, assuming that *mk_var* associates a unique CLP($\mathcal{R}$) variable (without any type annotation about variables being logical or arithmetic) to an arbitrary location in *DATAAREA* or *SVAR_AREA*:

$$\begin{array}{rcl} deref(l) & = & \begin{cases} deref(ref(l)) & \text{if } tag(l) = L\_var \ \&\ NOT(l\_unbound(l)) \\ l & \text{otherwise} \end{cases} \\ lpf(l) & = & c_0 + c_1 * svar(l_1) + \ldots + c_n * svar(l_n) \end{array}$$

$$term(l) = \begin{cases} mk\_var(l) & \text{if } l\_unbound(l) \\ mk\_var(ref(l)) & \text{if } s\_unbound(l) \\ term(deref(l)) & \text{if } tag(l) = L\_var \ \& \\ & \quad \text{NOT}(l\_unbound(\text{l})) \\ ref(l) & \text{if } tag(l) \in \{Const, Numb\} \\ svar(ref(l)) & \text{if } NOT(s\_unbound(l)) \ \& \\ & \quad tag(l) = S\_var \\ [\, term(ref(l)) \mid & \\ \quad term(ref(l)+) \,] & \text{if } tag(l) = List \\ f(a_1, \ldots, a_n) & \text{if } tag(l) = Struct \ \& \\ & ref(ref(l)) = \langle f, n \rangle \ \& \\ & \ term(ref(l)+i) = a_i \end{cases}$$

where $pfval(l)$ type is "=", $pfval(l+2) = n$, $pfval(l+4) = \langle c_0, nil \rangle$, $pfval(l+4+i) = \langle c_i, l_i \rangle$ for $1 \leq i \leq n$.

$$svar(l) = \begin{cases} mk\_var(l) & if s\_unbound(l) \\ lpf(svar\_lpf(ref(l))) & \text{if } s\_unfree(l) \\ svar\_value(ref(l)) & \text{if } s\_ground(l) \end{cases}$$

¿From these functions one can reconstruct arithmetic constraints by means of functions $eq, ineq, nlpf : PF\_AREA \rightarrow CONSTRAINT$ defined on locations l when the type of $pfval(l)$ is $=, \geq$ or $>$ and *nonlinear* respectively.

5 Unification

The "arithmetic part" of the current constraint system consists of all active equations, active inequalities and nonlinear constraints; the "logical part" is defined by the bindings in *DATAAREA*. The structures occurring during unification are represented as in Prolog on the stack usually called $(HEAP; h, boh)$, which is contained in *DATAAREA* and contains a 0-ary function *str*, a subterm (or structure) pointer to be used for navigating through substructures. The active part of *HEAP* will be abbreviated as $heap \equiv \{l \in HEAP \mid boh \leq l < h\}$ (its finiteness follows from *boh* being the initial value of h).

For unification we use the standard *pushdown list* stack

$$(PDL; pdl, nil; +, -)$$

with content function *pval* into *DATAAREA*. In the unification algorithm we add to the abstract update $bind(l_1, l_2)$ (to bind the "logical" variable stored at l_1 to $term(l_2)$) an update $equate(l_1, l_2)$ to call the equality solver for the two arithmetic terms stored at l_1 and l_2; see below. $bind(l_1, l_2)$, a "logical operation", is always consistent with the current collection of arithmetic constraints.

To the assumptions for *bind* in [4] we add an assumption for *equate*:

Bind Assumption. Let $l_1, l_2 \in \mathit{DATAAREA}$ and cs be the current constraint system. If $\mathit{l\_unbound}(l_1)$ holds and $\mathit{mk\_var}(l_1)$ does not occur in $\mathit{term}(l_2)$, then, after execution of $\mathit{bind}(l_1, l_2)$, the new constraint system is equal to

$$cs \bigcup \{\mathit{mk\_var}(l_1) = \mathit{term}(l_2)\}$$

If the new constraint system is unsolvable, then execution backtracks.

Equate Assumption. Let $l_1, l_2 \in \mathit{DATAAREA}$ such that $\mathit{term}(l_1)$ and $\mathit{term}(l_2)$ are "arithmetic terms", and cs be the current constraint system. After the execution of $\mathit{equate}(l_1, l_2)$, the new constraint system is the normalization[5] of

$$cs \bigcup \{\mathit{term}(l_1) = \mathit{term}(l_2)\}$$

If the new constraint system is unsolvable, then execution backtracks.

Unification will be triggered by setting a 0-ary function *What_to_do* with values in $\{\mathit{Run}, \mathit{Unify}\}$, given that the terms to be unified have already been pushed to *PDL*. The rules which control unification considerably extend the rules known for Prolog and can be obtained from Tables 1, 2 and 3.

The mathematical unification fails on an attempt to bind a variable to a term in which it occurs. To model such unification with "occur-check" it suffices to require of $\mathit{bind}(l_1, l_2)$ to trigger backtracking whenever $\mathit{mk\_var}(l_1)$ occurs in $\mathit{term}(l_2)$. In accordance with usual practice in logic programming, we do not specify the behavior of the system when the occur check fails.

The abbreviations in the unification tables are defined as follows; the contents of the empty slots can be obtained by the symmetric cases (with *dl* and *dr* exchanged):

$$\begin{aligned}
\mathit{left} &\equiv \mathit{pval}(\mathit{pdl}), \mathit{right} \equiv \mathit{pval}(\mathit{pdl}-), \mathit{dl} \equiv \mathit{deref}(\mathit{left}),\\
&\ \mathit{dr} \equiv \mathit{deref}(\mathit{right})\\
\mathit{failure} &\equiv \mathit{What\_to\_do} := \mathit{Run},\ \mathit{backtrack}\\
\mathit{top\_check} &\equiv \textbf{if } \mathit{value}(\mathit{dl}) \neq \mathit{value}(\mathit{dr}) \textbf{ then } \mathit{failure} \textbf{ else } \mathit{pdl} := \mathit{pdl}--
\end{aligned}$$

where $\mathit{value}(l)$ gives $\mathit{ref}(l)$ if l is a *Numb* cell; otherwise $\mathit{svar\_value}(\mathit{ref}(l))$. *list_check* denotes the updates for unification with lists $[l \mid l']$:

$$\begin{aligned}
\mathit{list\_check} \equiv\ & \mathit{pval}(\mathit{pdl}-) := \mathit{ref}(\mathit{dr}),\\
& \mathit{pval}(\mathit{pdl}) := \mathit{ref}(\mathit{dl}),\\
& \mathit{pval}(\mathit{pdl}+) := \mathit{ref}(\mathit{dr})+,\\
& \mathit{pval}(\mathit{pdl}++) := \mathit{ref}(\mathit{dl})+,\\
& \mathit{pdl} := \mathit{pdl}++
\end{aligned}$$

[5]See next section.

Table 1 Unification Table I.

	l_unbound(dl)	*s_unbound*(dl)	$tag(dl) =$ **Numb** or $s\_ground(dl)$
l_unbound(dr)	$bind(dl, dr)$ $pdl := pdl--$		
s_unbound(dr)	$bind(dl, dr)$ $pdl := pdl--$	$equate(dl, dr)$ $pdl := pdl--$	
$tag(dr) =$ **Numb** or $s\_ground(dr)$	$bind(dl, dr)$ $pdl := pdl--$	$equate(dl, dr)$ $pdl := pdl--$	*top_check*
tag(dr)=**Struct**	$bind(dl, dr)$ $pdl := pdl--$	*failure*	*failure*
s_unfree(dr)	$bind(dl, dr)$ $pdl := pdl--$	$equate(dl, dr)$ $pdl := pdl--$	$equate(dl, dr)$ $pdl := pdl--$
tag(dr)=**Const**	$bind(dl, dr)$ $pdl := pdl--$	*failure*	*failure*
tag(dr)=**List**	$bind(dl, dr)$ $pdl := pdl--$	*failure*	*failure*

Table 2 Unification Table II.

	$tag(dl) =$ **Struct**	**s_unfree**(dl)
$tag(dr) =$ **Struct**	*func_check*	
s_unfree(dr)	*failure*	$equate(dl, dr)$ $pdl := pdl--$
$tag(dr) =$ **Const**	*failure*	*failure*
$tag(dr) =$ **List**	*failure*	*failure*

func_check describes the updates for unification with terms $f(t_1, \ldots, t_n)$:

```
func_check  ≡  if $ref(ref(dl)) = ref(ref(dr))$
               then seq $pdl := pdl--$
                        seq $i = 1 \ldots arity(ref(dl))$
                            $pval(pdl+) := ref(dr) + i$
                            $pval(pdl++) := ref(dl) + i$
                            $pdl := pdl++$
                        endseq
                    endseq
               else $failure$
```

The crucial update *equate*, which we keep abstract here, has to re-

Table 3 Unification Table III.

	$tag(dl)$ = **Const**	$tag(dl)$ =**List**
$tag(dr)$ = **Const**	*top_check*	
$tag(dr)$ = **List**	*failure*	*list_check*

flect the complex interaction between solver modules for linear equations, inequalities and nonlinear equations (see Fig. 3). In particular *equate* is assumed to:

1. Rewrite the equation $term(l_1) = term(l_2)$ in such a way that exactly one variable (called the subject) appears on the left-hand side[6]; the result is stored in *PF_AREA* and occurrences lists are updated.
2. "Propagate" the rewritten equation to other constraints so that occurrences lists are updated and constraints are mantained in solved form.
3. Invoke the Simplex algorithm if the subject of the new equation is contained in inequalities; the set of implied equalities is calculated and each of them is handled by *equate*.
4. Calculate the wakeup degrees of all modified nonlinear constraints for each grounding equation which affects nonlinear equalities; if a nonlinear constraint is awoken[7], then an *equate* update is invoked.

It is further assumed that whenever a linear form must be changed, it is put to sleep and rewritten on top of *PF_AREA*. ¿From the above assumptions one can prove the following:

Lemma 5.1. (Unification Lemma) *Let l_1, l_2 be in DATAAREA such that $term(l_1), term(l_2) \in TERM$ and let cs be the current constraint system. After execution of unify(l_1, l_2), the new constraint system is the normalization of*

$$cs \bigcup \{term(l_1) = term(l_2)\}$$

If the new constraint system is unsolvable, then execution backtracks.

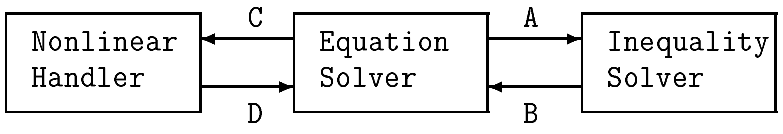
6 Compilation of Constraints

In this section we show how the formal description of WAM compilation of terms can be naturally extended to constraints. For simplicity we assume here that all variables are permanent and get initialized to *unbound* as soon as they are allocated[8].

[6]The choice of the subject variable is ruled by efficiency criteria.

[7]An awoken nonlinear constraint gives rise to a linear equation.

[8]This assumption can be eliminated later by a refined variable classification.



```
A: Equation affecting inequalities;
B: Inferred equality;
C: Equation affecting nonlinear equalities;
D: Awoken nonlinear constraint.
```

FIG. 3. Interactions between solver modules.

Term arguments occurring during term compilation will be represented in a register subdomain *AREGS* of *DATAAREA*, disjoint from the heap. Registers are numbered by a function $x : \mathcal{N} \rightarrow AREGS$ for which we write $x_i \equiv x(i)$.

The CLAM code for compiled body subgoals is accessed from a subset *CODEAREA* of *MEMORY* through a content function *code* into the universe *INSTR* of instructions which is assumed to contain all WAM instructions and the following instructions for handling of constraints:

$unify\_number(c)$	$put\_number(c, x_j)$	$get\_number(c, x_j)$
$initpf(c)$	$addpf\_val(c, x_j)$	$addpf\_val(c, y_n)$
$addpf\_var(c, x_j)$	$get\_lpf(x_j)$	$set\_svariable(x_j)$
$solve\_eq0$	$solve\_ge0$	$solve\_gt0$
$put\_lpf(x_j)$		

with $n, j \in \mathcal{N}$, $y_n \in DATAAREA$, $c \in NUMBER$, $x_j \in AREGS$ (*INSTR* will be tacitly extended with further instructions occurring below).

For compilation of arithmetic constraints, coming as streams of simple token patterns, we need a concept of constraint normal form which is an analogue of the logical notion of term normal form. An arithmetic equation, inequality or nonlinear constraint is said to be in normal form if it is of form $lpf = 0$, $lpf \geq 0$, $lpf > 0$, $nlpf$, respectively, where *lpf* is a linear parametric form and *nlpf* is a nonlinear form.

We extend the usual normalization procedures nf_s and nf_a —corresponding to the analysis (for putting instructions) and synthesis (for getting instructions) of terms—from logical term equations to constraints as follows:

$$nf(X_i = Y_n) = [\,X_i = Y_n\,], \qquad nf(X_i = c) = [\,X_i = c\,]$$
$$nf(Y_i = Y_n) = nf(X_i = X_i) \quad = \quad nf(c = c) = [\;]$$

$$\begin{array}{rcl} \mathit{nf}_s(X_i = c_0 + c_1 s_1 + \ldots + c_n s_n) & = & \mathit{flatten}([\,\mathit{nf}_s(Z_1 = s_1), \ldots, \\ & & \qquad \mathit{nf}_s(Z_n = s_n), \\ & & X_i = c_0 + c_1 Z_1 + \ldots + c_n Z_n\,]) \end{array}$$

The same applies for linear inequalities replacing $X_i =$ by $0 <, 0 \leq$ respectively:

$$\begin{array}{rcl} \mathit{nf}_s(s_0 = \mathit{op}(s_1, \ldots, s_m)) & = & \mathit{flatten}([\,\mathit{nf}_s(Z_0 = s_0), \\ & & \mathit{nf}_s(Z_1 = s_1), \ldots, \mathit{nf}_s(Z_m = s_m), \\ & & \qquad Z_0 = \mathit{op}(Z_1, \ldots, Z_m)\,]) \\ \mathit{nf}_a(X_i = \mathit{op}(s_1, \ldots, s_m)) & = & \mathit{flatten}([X_i = \mathit{op}(Z_1, \ldots, Z_m), \\ & & \mathit{nf}_a(Z_1 = s_1), \ldots \mathit{nf}_a(Z_m = s_m)]) \\ \mathit{nf}_a(X_i = c_0 + c_1 s_1 + \ldots + c_n s_n) & = & \mathit{flatten}([X_i = c_0 + c_1 Z_1 + \ldots + \\ & & \qquad c_n Z_n, \\ & & \mathit{nf}_a(Z_1 = s_1), \ldots, \mathit{nf}_a(Z_n = s_n)]) \end{array}$$

where nf stands for both nf_s and nf_a; Z_i is equal to s_i if s_i is a constant or a variable, otherwise it is a fresh X variable.

Use of normalization is justified by the well known fact that the constraint $s\Delta t$ is equivalent to the set of constraints $\mathit{nf}(s\Delta t)$; computationally this is reflected by the CLP($\mathcal{R}$) effect of executing $s\Delta t$ being the same as executing all members of $\mathit{nf}(s\Delta t)$.

For each nonlinear operation, instructions are provided for creating a nonlinear constraint of any degree. For example, we could define five instructions for *pow*, corresponding to the various wakeup degrees (see [10]):

- $\mathit{pow\_vvv}(V_i, V_j, V_k)$ for $V_i = \mathit{pow}(V_j, V_k)$, $\mathit{pow\_cvv}(c, V_j, V_k)$ for $c = \mathit{pow}(V_j, V_k)$
- $\mathit{pow\_vcv}(V_i, c, V_k)$ for $V_i = \mathit{pow}(c, V_k)$, $\mathit{pow\_vvc}(V_i, V_j, c)$ for $V_i = \mathit{pow}(V_j, c)$
- $\mathit{pow\_cvc}(c_0, V_j, c_1)$ for $c_0 = \mathit{pow}(V_j, c_1)$

We suppose that we have an auxiliary function *nl_instr* that takes a normalized nonlinear constraint and returns the appropriate instruction. For example,

$$\mathit{nl\_instr}(X = \mathit{pow}(3, Y)) \quad = \quad \mathit{pow\_vcv}(X, 3, Y)$$

6.1 Putting Instructions (Body Subgoals Compilation)

We extend the WAM function *put_instr*, yielding the sequence of instructions which compiles by a normalized equation, to provide also compiled code for constraints. The extension is defined according to the following table, where j stands for an arbitrary "top level" index (corresponding to input $X_j = t$ for term normalization), k for a "non top level" index (corresponding to an auxiliary variable introduced by normalization itself) and i

for any index:

$$\begin{array}{rcl} X_j = c & \rightarrow & [\, \mathit{put\_number}(c, x_j)\,] \\ X_i = c_0 + c_1 Z_1 + \ldots + c_n Z_n & \rightarrow & [\mathit{initpf}(c_0), \mathit{addpf}(c_1, z_1), \ldots, \\ & & \quad \mathit{addpf}(c_n, z_n), \mathit{put\_lpf}(x_i)] \end{array}$$

where $\mathit{addpf}(c, z_i)$ is $\mathit{addpf\_val}(c, y_n)$ if $Z_i = Y_n$ and $\mathit{addpf\_val}(c, x_i)$ if $Z_i = X_i$. The same applies for $c_0 + c_1 Z_1 + \ldots + c_n Z_n \ \Delta\ 0$ replacing $\mathit{put\_lpf}(x_i)$ by $\mathit{instr}(\Delta)$, where

$$\mathit{instr}(\Delta) \quad = \quad \begin{cases} \mathit{solve\_eq0} & \text{if } \Delta = \text{``}=\text{''} \\ \mathit{solve\_ge0} & \text{if } \Delta = \text{``}\geq\text{''} \\ \mathit{solve\_gt0} & \text{if } \Delta = \text{``}>\text{''} \end{cases}$$

Next we consider the case of nonlinear constraint. Note the use of the *set_svariable* instruction because nonlinear instructions are supposed to handle variables which are already initialized:

$$\begin{array}{rcl} X_i = op(Z_1, \ldots, Z_m) & \rightarrow & [\mathit{set\_svariable}(x_i), \mathit{nl\_instr}(X_i = op(Z_1, \ldots, Z_m))] \\ V = op(Z_1, \ldots, Z_m) & \rightarrow & [\mathit{nl\_instr}(V = op(Z_1, \ldots, Z_m))] \\ & & \text{where } V = c \text{ or } V = Y_n \end{array}$$

Moreover, a numeric constant c appearing inside a structure or a list is compiled into the instruction *unify_number*(c), with $y_n \in \mathit{DATAAREA}$, $x_i \in \mathit{AREGS}$.

The function *put_seq*—defining the compilation of a body goal, see [4]—is here extended to specify how constraints are to be compiled. The definition uses the auxiliary function *put_code*, defined by flattening the result of mapping *put_instr* along $nf_s(X_i = t)$, $nf_s(c_0 + c_1 s_1 + \ldots + c_n s_n \ \Delta\ 0)$ and $nf_s(s_0 = op(s_1, \ldots, s_m))$, where $\Delta \in \{=, >, \geq\}$. For terms s_i where the outermost function symbol is not arithmetic, we set as for body goals in the WAM:

$$\begin{array}{rcl} \mathit{put\_seq}(s_1 = s_2) & = & \mathit{flatten}([\mathit{put\_code}(X_1 = s_1), \mathit{put\_code}(X_2 = s_2)]) \\ & & \text{with top level } j = 1, 2 \end{array}$$

For nonlinear constraints *put_seq* coincides with *put_code*:

$$\mathit{put\_seq}(s_0 = op(s_1, \ldots, s_m)) \quad = \quad \mathit{put\_code}(s_0 = op(s_1, \ldots, s_m))$$

On arithmetic constraints, *put_seq* applies *put_code* to a simplified constraint version where all terms with the same variable are collected and all constants are added:

$$\begin{array}{rcl} \mathit{put\_seq}(t_1 = t_2) & = & \mathit{put\_code}(\mathit{simplify}(t_1 - t_2 = 0)) \\ \mathit{put\_seq}(t_1 > t_2) & = & \mathit{put\_code}(\mathit{simplify}(t_1 - t_2 > 0)) \\ \mathit{put\_seq}(t_1 \geq t_2) & = & \mathit{put\_code}(\mathit{simplify}(t_1 - t_2 \geq 0)) \end{array}$$

$$\begin{aligned} put\_seq(t_1 < t_2) &= put\_code(simplify(t_2 - t_1 > 0)) \\ put\_seq(t_1 \leq t_2) &= put\_code(simplify(t_2 - t_1 \geq 0)) \end{aligned}$$

In the following rules which describe the execution of code generated by putting, an auxiliary stack ($ACCUMULATOR; acctop, accbottom$) is needed which allows us to construct linear forms to build arithmetic constraints. Its content function *accval* has to yield constants or pairs (constant, solver identifier).

Also a 0-ary function *add_counter* is added containing the number of parametric variables which have appeared so far in the linear form being read.

For the WAM putting instructions the rules defined in [4] are taken. For the CLAM instructions the rules are defined now. The rules for *put_number* and *unify_number* are like those for WAM instructions *put_constant* and *unify_constant*:

```
if $code(p) = put\_number(c, x_j)$
then $x_j \leftarrow \langle Numb, c \rangle$
     $succeed$
```

```
if $code(p) = unify\_number(c)$
   & $mode = Write$
then $h \leftarrow \langle Numb, c \rangle$
     $h := h+$
     $succeed$
```

The rule for $set\_svariable(x_i)$ is used for initializing a fresh free solver variable pointed at by x_i. It is similar to the instruction *set_variable* used in [1]:

```
if $code(p) = set\_svariable(x_i)$
then $mk\_s\_unbound(x_i)$
     $succeed$
```

The instruction *set_svariable* will be optimized away in the sequel.

The rule that begins the construction of a linear parametric form[9] puts the constant on top of the accumulator and sets *add_counter* to zero (there are still no parametric variables):

```
if $code(p) = initpf(c)$
then $accval(acctop) := c$
     $acctop := acctop+$
     $add\_counter := 0$
     $succeed$
```

The instruction $addpf\_var(c, l)$ enriches the linear form being built in

[9]Note that the arithmetic term which is "accumulated" will not necessarily result in a linear form: it may be simply a numerical constant.

the accumulator with a variable to be initialized:

```
**if** $code(p) = addpf\_var(c, x_i)$
**then** $mk\_s\_unbound(x_i)$
     $accval(acctop) \leftarrow \langle c, svtop \rangle$
     $acctop := acctop+$
     $add\_counter := add\_counter + 1$
     $succeed$
```

The instruction $addpf\_val(c, l)$ enriches the linear form being built in the accumulator with a new component. We have the following cases:

1. $deref(l)$ gives the address of a *Numb* cell or a ground solver variable: the constant so obtained is multiplied by c and then the result is added to the constant already stored at the bottom of the accumulator.
2. $deref(l)$ gives the address of a free logical variable: the logical variable is transformed into a solver variable; its identifier is put onto the accumulator (along with the constant c) and *add_counter* is incremented by 1.
3. $deref(l)$ gives the address of a free solver variable: if the solver variable is a parametric variable already appearing in the linear form being built with coefficient c', then c is added to c'; otherwise the element $\langle c, ref(deref(l)) \rangle$ is put onto the accumulator and *add_counter* is incremented by 1.
4. $deref(l)$ gives the address of an unfree solver variable pointing to a linear form *lpf*: the term to be added is $c * lpf$.

Note that the linear form is built so as to be in solved form when it has to be copied in *PF_AREA*. In the *addpf_val* rule we make use of an abstract binary update *add*, for which we assume that it does what is described in (3) and (4):

$$add(c') \equiv accval(accbottom) := accval(accbottom) + c * c'$$

and of the trail update discussed below:

```
if $code(p) = addpf\_val(c,l)$ then case $tag(deref(l))$ of
  $Numb$  :  $add(ref(deref(l)))$
             $succeed$
  $L\_var$ :  $mk\_s\_unbound(deref(l))$
             $trail(deref(l))$
             $accval(acctop) \leftarrow \langle c, svtop\rangle$
             $acctop := acctop +$
             $add\_counter := add\_counter + 1$
             $succeed$
  $S\_var$ :  if $s\_ground(deref(l))$
             then $add(svar\_value(ref(deref(l))))$
                  $succeed$
             else $add(c, deref(l))$
                  $succeed$
  $other$ :  $backtrack$
```

The instruction $put\_lpf(x_i)$ concludes the construction of a linear form by storing the equation in memory. We have two possible cases:

1. $add\_counter = 0$. The arithmetical term built on the accumulator is a numerical constant: in this case register x_i is instantiated to that constant.
2. $add\_counter > 0$: the arithmetic term built on the accumulator is actually a linear form that has to be stored on top of PF_AREA. A new free solver identifier is initialized and then the equality solver is called:

```
if $code(p) = put\_lpf(x_i)$
then $succeed$
     if $add\_counter = 0$ then $x_i \leftarrow \langle NUMB, accval(accbottom)\rangle$
                          else $mk\_s\_unbound(x_i)$
                               $trigger\_equate(x_i, accbottom)$
```

In the following we make use of an abbreviation to store linear inequalities:

```
$store(type)$  $\equiv$  $pfval(pftop) := type$
                  $pfval(pftop+) := active$
                  $pfval(pftop + 2) := add\_counter$
                  $pfval(pftop + 3) := \langle accval(accbottom), nil\rangle$
                  seq $j = 1, \ldots, add\_counter$
                    $pfval(pftop + 3 + j) := accval(accbottom + j)$
                  endseq
                  $pftop := pftop + add\_counter + 4$
```

where *trigger_equate* is an abstract update which triggers *equate*[10]. The instruction *put_lpf* will become superfluous in the sequel.

The instruction *solve_ge0* concludes the construction of an arithmetic term in the accumulator; this term must be compared with zero if it is ground, otherwise it must be stored as a nonstrict inequality in *PF_AREA*. We have two cases:

1. *add_counter* $= 0$: the contructed arithmetic term is a numeric constant, so the test can be done;
2. *add_counter* > 0: the contructed arithmetic term is a linear form, so it has to be copied onto *PF_AREA* with type $\geq$; Lists of occurrences are updated (using an abstract update *update_occ*) and the inequality solver is called.

The call to the inequality solver is represented by an abstract update *inequality*(l), which yields *backtrack* if the Simplex tableau becomes unsolvable. Eventual equalities are inferred and for each of them the equality solver is called.

The same also holds for *solve_gt0* which generates a strict inequality:

```
if $code(p) = solve\_ge0 \ \mid \ solve\_gt0$
then if $add\_counter = 0$
     then if $accval(accbottom) \geq 0 \ \mid > 0$
          then $succeed$
          else $backtrack$
     else $store(\geq) \ \mid \ store(>)$
          $update\_occ$
          $top_{\geq} := pftop \ \mid \ top_{>} := pftop$
          $succeed$
          $trigger\_inequality(pftop)$
```

where *trigger_inequality* is an abstract update which triggers the update *inequality*.

The rule for *solve_eq0* is quite similar to the previous one:

```
if $code(p) = solve\_eq0$
then if $add\_counter = 0$
     then if $accval(accbottom) = 0$
          then $succeed$
          else $backtrack$
     else $succeed$
          $trigger\_equate(accbottom, 0)$
```

[10]Note that the equality solver is called after the creation of the linear form and the solver variable.

For instructions for nonlinear constraints, we consider as a typical example the real operator *sin*, defined for any real number (so it is not invertible). Sleeping nonlinear constraints involving this function have only one wakeup degree, call it *ind*, and are awoken when the argument gets instantiated to a ground term. *sin* has two instructions $sin\_vv(l_1, l_2)$ and $sin\_cv(c, l)$. Using the abbreviation

```
$store\_nlpf(l)$  $\equiv$  $pfval(pftop) := nonlinear$
                   $pfval(pftop+) := \langle sin, ind, 1\rangle$
                   $pfval(pftop + 2) := c$
                   $pfval(pftop + 3) := l$
                   $nleq := pftop$
                   $pftop := pftop + 4$
                   $update\_occ$
```

we have the rule for *sin_cv*, using *dl* instead of *deref*(*l*):

```
if $code(p) = sin\_cv(c, l)$
then case $tag(dl)$ of
  $Numb$  :  if $c = sin(ref(dl))$
             then $succeed$
             else $backtrack$
  $L\_var$ :  if $-1 \leq c \leq 1$
             then $trail(dl)$
                  $mk\_s\_unbound(dl)$
                  $store\_nlpf(svtop)$
                  $succeed$
             else $backtrack$
  $S\_var$ :  if $s\_ground(dl)$
             then if $c = sin(svar\_value(ref(dl)))$
                  then $succeed$
                  else $backtrack$
             else if $-1 \leq c \leq 1$
                  then $store\_nlpf(ref(dl))$
                       $succeed$
                  else $backtrack$
  $other$ :  $backtrack$
```

¿From the assumptions made for the abstract updates introduced above one can prove:

Lemma 6.1. (Putting Lemma I) *Let* $\{Y_1, \ldots Y_l\ \}$ *be the set of variables occurring in the CLP(*$\mathcal{R}$*) literal* $g(t_1, \ldots, t_n)$*,* $y_n \in DATAAREA$ *with* $term(y_n) \in TERM$ $(n = 1, \ldots, l)$*, cs be the current constraint system associating every* Y_n *with* $term(y_n)$ *and* X_i *be fresh pairwise distinct variables,* $i = 1, \ldots, m$*. The effect of executing (setting p to a value where* *unload yields) put_seq($g(t_1,\ldots,t_n)$) is that the new constraint system is the normalization of*

$$cs \bigcup \{X_i = t_i\}$$

Lemma 6.2. (Putting Lemma II) *Let $\{Y_1,\ldots,Y_l\}$ be all the variables occurring in the CLP($\mathcal{R}$) constraint c, $y_n \in$ DATAAREA with $term(y_n) \in$ TERM ($n = 1,\ldots,l$) and let cs be the current constraint system associating Y_n with $term(y_n)$. The effect of executing put_seq(c) is that if $cs \bigcup \{c\}$ is solvable then it will be equal (up to normalization) to the new constraint system; otherwise execution backtracks.*

6.2 Getting Instructions (Clause Head Compilation)

The compilation of clause heads is defined in the same way as clause body compilation where only the function *get_instr* is different.

For $X_j = c$ and $X_i = c_0 + c_1 Z_1 + \ldots + c_n Z_n$ it suffices to replace *put* by *get*, whereas for nonlinear forms we set

$$\begin{array}{lcl} X_i = op(Z_1,\ldots,Z_m) & \to & flatten([set\_seq([Z_1,\ldots,Z_m]), \\ & & nl\_instr(X_i = op(Z_1,\ldots,Z_m))]) \end{array}$$

The *set_seq* ensure that when a variable X_i first occurs in a nonlinear constraint, it is initialized before executing the appropriate "nonlinear" instruction[11]:

$$\begin{array}{rcl} set\_seq([]) & = & [] \\ set\_seq([X_i \mid T]) & = & [set\_svariable(x_i) \mid set\_seq(T)] \\ set\_seq([c \mid T]) & = & set\_seq(T) \\ set\_seq([Y_n \mid T]) & = & set\_seq(T) \end{array}$$

In addition to the rules for WAM's getting instructions (see [4]), we define now the rules for new CLAM instructions. The *get_number* rule is

[11]Use of *set_seq* could been avoided by introducing variants of nonlinear instructions which allow a variable to be initialized. For example, for a constraint $c = sin(Y_n)$ where Y_n appears in its first occurrence, the instruction *sin_cV* can be used with execution rule:

```
if $code(p) = sin\_cV(c,l)$ then if $-1 \leq c \leq 1$ then $mk\_s\_unbound(l)$
                                                   $store\_nlpf(svtop)$
                                                   $succeed$
                                              else $backtrack$
```

It is easy to see that the sequence $set\_svariable(y_n)$, $sin\_cv(c,y_n)$ is equivalent to $sin\_cV(c,y_n)$.

analogous to the *get_constant* rule:

```
if $code(p) = get\_number(c, x_i)$
then case $tag(deref(x_i))$ of
 $Numb$   :  if $c = ref(deref(x_i))$
             then $succeed$
             else $backtrack$
 $L\_var$ :  $deref(x_i) := \langle Numb, c\rangle$
             $trail(deref(x_i))$
             $succeed$
 $S\_var$ :  if $s\_ground(deref(x_i))$
             then  if $c = svar\_value(ref(deref(x_i)))$
                   then $succeed$
                   else $backtrack$
             else  $succeed$
                   $trigger\_equate(deref(x_i), c)$
 $other$  :  $backtrack$
```

The rule for *unify_number* is analogous to the rule for *unify_constant*:

```
if  $code(p) = unify\_number(c)$ & $mode = Read$
then case $tag(deref(str))$ of
 $Numb$   :  if $c = ref(deref(str))$
             then  $str := str+$
                   $succeed$
             else $backtrack$
 $L\_var$ :  $deref(str) := \langle Numb, c\rangle$
             $trail(deref(str))$
             $str := str+$
             $succeed$
 $S\_var$ :  if $s\_ground(deref(str))$
             then  if $c = svar\_value(ref(deref(str)))$
                   then  $str := str+$
                         $succeed$
                   else $backtrack$
             else  $str := str+$
                   $succeed$
                   $trigger\_equate(deref(str), c)$
 $other$  :  $backtrack$
```

The instruction $get\_lpf(x_i)$ unifies the arithmetic term built in the accumulator with $term(x_i)$. In order to make updates more readable we split the rule into two. The first regards the case in which the term constructed in the accumulator is a constant:

```
if  $code(p) = get\_lpf(x_i)$ & $add\_counter = 0$
then case $tag(deref(x_i))$ of
  $Numb$   :  if $ref(deref(x_i)) = accval(accbottom)$
              then $succeed$
              else $backtrack$
  $L\_var$ :  $deref(x_i) \leftarrow \langle Numb, accval(accbottom)\rangle$
              $trail(deref(x_i))$
              $succeed$
  $S\_var$ :  if $s\_ground(deref(x_i))$
              then if $accval(accbottom) = svar\_value(ref(deref(x_i)))$
                     then $succeed$
                     else $backtrack$
              else $succeed$
                     $trigger\_equate(deref(x_i), accbottom)$
  $other$  :  $backtrack$
```

The second rule regards the case in which a linear form has been built:

```
if  $code(p) = get\_lpf(x_i)$ & $add\_counter > 0$
then case $tag(deref(x_i))$ of
  $Numb, S\_var$ :  $succeed$
                    $trigger\_equate(deref(x_i), accbottom)$
        $L\_var$ :  $mk\_s\_unbound(deref(x_i))$
                    $trail(deref(x_i))$
                    $succeed$
                    $trigger\_equate(deref(x_i), accbottom)$
        $other$  :  $backtrack$
```

The instruction *get_lpf* will be optimized away in the sequel. For the getting code described above one can prove:

Lemma 6.3. (Getting Lemma) *Let all variables occurring in the literal $g(t_1, \ldots, t_m)$ be among $\{Y_1, \ldots, Y_l\}$, and let further $y_n \in DATAAREA$ with $l\_unbound(y_n)$, $n = 1, \ldots, l$, X_i fresh pairwise distinct variables with $x_i \in DATAAREA$ and $term(x_i) \in TERM$, $1 \leq i \leq m$. Let cs be the current constraint system. If*

$$cs \bigcup \{term(x_i) = t_i\}$$

is solvable, then, after executing (setting p to) $get\_seq(g(t_1, \ldots, t_m))$, it will be equal (up to normalization) to the new constraint system; otherwise execution backtracks.

Modifying *put-* and *get-code* (to generate *unify_local_value* instead of *unify_value* for all occurrences of local variables), as in the WAM one can

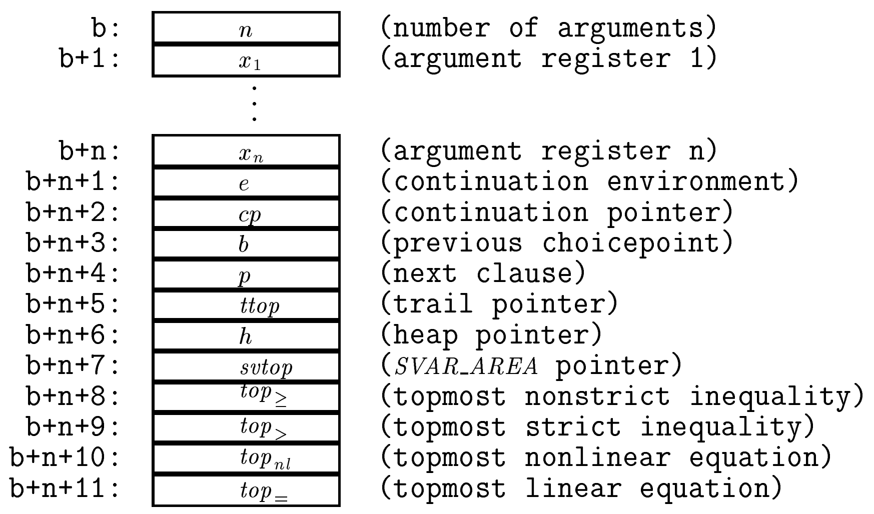

b: n (number of arguments)
b+1: x_1 (argument register 1)
b+n: x_n (argument register n)
b+n+1: e (continuation environment)
b+n+2: cp (continuation pointer)
b+n+3: b (previous choicepoint)
b+n+4: p (next clause)
b+n+5: $ttop$ (trail pointer)
b+n+6: h (heap pointer)
b+n+7: $svtop$ (*SVAR_AREA* pointer)
b+n+8: $top_{\geq}$ (topmost nonstrict inequality)
b+n+9: $top_{>}$ (topmost strict inequality)
b+n+10: top_{nl} (topmost nonlinear equation)
b+n+11: $top_{=}$ (topmost linear equation)

FIG. 4. Choicepoint frame.

preserve the *Heap Variable Constraint* (that no heap variable points outside the heap); see [4].

7 CLAM

In this section we obtain a description of the CLAM by embedding the compilation of terms and constraints into the CLP($\mathcal{R}$) model with predicate and clause compilation (outlined in section 3).

7.1 Stack and Trail

The first step is, as in the WAM, to refine the choicepoint and environment stack to become a subalgebra $(STACK; tos(b,e), bos; +, -; val)$ of *DATAAREA* disjoint from *HEAP* and *AREGS*. The distinguished elements b and e stand for the (address of the) topmost choicepoint and environment, respectively. The environment frame is taken unchanged from the WAM (see [4]), while the choicepoint frame gets extended by information on constraints as described in Fig. 4. Formally:

$$\begin{array}{lll} svtop(l) \equiv l+n+7 & top_{\geq}(l) \equiv l+n+8 & top_{>}(l) \equiv l+n+9 \\ top_{nl}(l) \equiv l+n+10 & top_{=}(l) \equiv l+n+11 & hb \equiv val(h(b)) \\ n(l) \equiv l & & \end{array}$$

Thus we have the same form of environment (de-)allocation and choicepoint handling rules as for the WAM (including the conditions on binding and $HEAP < STACK < AREGS$; see [4]). We have to refine in particular the *trail* update, which allows to undo, on backtracking, all the modifications to the constraint system made after creation of the choicepoint where the system backtracks to. The stack $(TRAIL, TO; ttop, tbottom; +, -)$ is thus extended to contain (via a function *tval*) not only *DATAAREA* locations, but *trail objects* which are tagged and refer to the items to be stored:

$$\begin{array}{lcll} ttag & : & TO \to & \{FREE\_L, FREE\_S, S\_var, LPF, ID, NLPF\} \\ tref & : & TO \to & DATAAREA + PF\_AREA + \\ & & & (PF\_AREA \times SVAR\_AREA) + \\ & & & + SVAR\_AREA + (PF\_AREA \times \mathcal{N}) \end{array}$$

under the following conditions (writing $ttag(l)$ for $ttag(tval(l))$, $tref(l)$ for $tref(tval(l))$):

$$\begin{array}{rcl} \textbf{case} & ttag(l) & \textbf{of} \\ FREE\_L & : & tref(l) \in DATAAREA \\ FREE\_S & : & tref(l) \in SVAR\_AREA \\ ID, S\_var & : & tref(l) \in (PF\_AREA \times SVAR\_AREA) \\ LPF & : & tref(l) \in PF\_AREA \\ NLPF & : & tref(l) \in (PF\_AREA \times \mathcal{N}) \end{array}$$

According to the location to be trailed the update $trail(l)$, assumed to be executed at each invocation of *bind* or *equate* concerning l, can be:

- $trail\_lf(l)$ if $l\_unbound(l)$; l is an unbound logical variable;
- $trail\_sf(l)$ if $id\_unbound(l)$; l is an unbound solver identifier;
- $trail\_sid(l)$ if $id\_unfree(l)$; l is a solver identifier equated to a linear form;
- $trail\_lpf(l)$ if $pfval(l) \in \{=, \geq, >\}$; l is the initial address of a linear form which is going to be put to sleep;
- $trail\_nlpf(l)$ if $pfval(l) = nonlinear$; l is the address of a nonlinear constraint whose wakeup degree is going to be modified[12];
- $trail\_id(l)$ if $pfval(l) \in SVAR\_AREA$; l contains the address of a non-constant parameter of a nonlinear constraint.

$$\begin{array}{rcl} trail\_lf(l) & \equiv & ttag(ttop) := FREE\_L, \\ & & tref(ttop) := l, \\ & & ttop := ttop+ \\ trail\_sf(l) & \equiv & ttag(ttop) := FREE\_S, \\ & & tref(ttop) := l, \\ & & ttop := ttop+ \end{array}$$

[12]Alternatively, the wakeup degree of all nonlinear constraints affected by trailing operations could be calculated as part of the *backtrack* update.

$$\begin{aligned}
\mathit{trail\_sid}(l) \quad &\equiv \quad \mathit{ttag}(\mathit{ttop}) := \mathit{S\_var},\\
&\qquad \mathit{fst}(\mathit{ttref}(\mathit{ttop})) := \mathit{svar\_lpf}(l),\\
&\qquad \mathit{snd}(\mathit{tref}(\mathit{ttop})) := l,\\
&\qquad \mathit{ttop} := \mathit{ttop}+\\
\mathit{trail\_lpf}(l) \quad &\equiv \quad \mathit{ttag}(\mathit{ttop}) := \mathit{LPF},\\
&\qquad \mathit{tref}(\mathit{ttop}) := l,\\
&\qquad \mathit{ttop} := \mathit{ttop}+\\
\mathit{trail\_nlpf}(l) \quad &\equiv \quad \mathit{ttag}(\mathit{ttop}) := \mathit{NLPF},\\
&\qquad \mathit{fst}(\mathit{tref}(\mathit{ttop})) := l,\\
&\qquad \mathit{snd}(\mathit{tref}(\mathit{ttop})) := \mathit{third}(\mathit{pfval}(l+)),\\
&\qquad \mathit{ttop} := \mathit{ttop}+\\
\mathit{trail\_id}(l) \quad &\equiv \quad \mathit{ttag}(\mathit{ttop}) := \mathit{ID},\\
&\qquad \mathit{fst}(\mathit{tref}(\mathit{ttop})) := l,\\
&\qquad \mathit{snd}(\mathit{tref}(\mathit{ttop})) := \mathit{pfval}(l),\\
&\qquad \mathit{ttop} := \mathit{ttop}+
\end{aligned}$$

Accordingly, the WAM backtrack update must be refined to undo all constraints trailed after the current choicepoint was pushed. In the following definition *ttop*(b) yields the value *ttop* had when the choicepoint b was created. Here is the definition:

```
backtrack  ≡  **if** $b = bos$ **then** $stop := -1$
              **else** $p := val(p(b))$
                    **seq** $l = ttop - \ldots tr(b)$
                      **case** $ttag(l)$ **of**
                          $FREE\_L$  :  $mk\_l\_unbound(tref(l))$
                          $FREE\_S$  :  $mk\_id\_unbound(tref(l))$
                          $S\_var$   :  $svar\_lpf(snd(tref(l))) :=$
                                              $fst(tref(l))$
                                        $update\_occ$
                          $LPF$      :  $pfval(tref(l)+) := active$
                                        $update\_occ$
                          $NLPF$     :  $third(pfval(fst(tref(l)))) :=$
                                                $snd(tref(l))$
                          $ID$       :  $pfval(fst(tref(l))) :=$
                                            $snd(tref(l))$
                                        $update\_occ$
                      **endcase**
                    **endseq**
```

(Recall that we assume that *update_occ* adjusts occurrence lists.)

Trailing operations (and backtracking) also have to take into account the result of the inequality solver. Since we keep the latter unspecified, we assume that trailing records the inequality solver data structures (for example the Simplex tableau) which are relevant for the currently "active" inequalities stored in *PF_AREA*.

Putting together all the assumptions made for compilation of predicates, clauses, terms and constraints one can prove the main theorem stated in the introduction; see [11].

7.2 Optimizations

In this section we outline some of the major optimizations for the CLAM.

The classification of variables—which is crucial for environment trimming and last call optimization—can be taken together with all the proofs almost literally from the WAM model of [4] thanks to the modularity of the specification introduced here. We state here only the adaptation, to clauses with constraints, of the definition of permanent and temporary variables:

Definition 7.1. *A variable occurring before or in a body subgoal α_i of a clause $H :- \alpha_1, \ldots, \alpha_n$, is needed at α_i, $1 \leq i < n$, if either (a) it occurs in some α_j, $j > i$, and there exists k with $i \leq k < j$ such that α_k is an atom or (b) α_i is an atom (not the last one) and the first occurrence of the variable in the clause is an argument position of α_i.*

Definition 7.2. *A variable appearing in a clause is said to be permanent if it is needed at some body subgoal; otherwise it is temporary.*

The environment allocation is superfluous when the clause body is empty or contains only constraints or contains only constraints followed by one atom: in all these cases no variable is needed at any subgoal.

Trailing operations on variables can be optimized by executing the update $\mathit{trail\_lf}(l)$ only if ($l \in \mathit{heap}$ & $l < hb$) or ($l \in \mathit{STACK}$ & $l < b$) just as in the WAM. An optimization for linear parametric forms can be obtained by executing $\mathit{trail\_lpf}(l)$ only if one of the following conditions holds:

$$\begin{array}{lcl} \mathit{pfval}(l) = eq & \& & \mathit{val}(\mathit{leq}(b)) \geq l \\ \mathit{pfval}(l) = ge & \& & \mathit{val}(\mathit{nsineq}(b)) \geq l \\ \mathit{pfval}(l) = gt & \& & \mathit{val}(\mathit{sineq}(b)) \geq l \end{array}$$

Also $\mathit{trail\_sf}(l)$ and $\mathit{trail\_sid}(l)$ can be optimized, executing them only if $l < \mathit{val}(\mathit{svtop}(b))$. For nonlinear constraints $\mathit{trail\_nlpf}(l)$ and $\mathit{trail\_id}(l)$ can be optimized by executing them only if $\mathit{val}(\mathit{nleq}(b)) \geq l$.

One of the principal mechanisms for enhancing efficiency is to avoid invoking the full solver when constraints are simple. As we have seen in section 5, the process of solving a linear constraint consists first of building a linear form in the accumulator[13] and then handling the constraint by means of *solve_eq0*, *solve_ge0* or *solve_gt0* instructions. Solving an equation amounts to:

[13] A linear form built on the accumulator contains only parametric and new variables.

1. finding a parameter V to become nonparametric;
2. writing the equation with subject V, i.e. in the form $V = lpf$;
3. substituting out V using *lpf* in all other linear constraints;
4. adding the new constraint $V = lpf$.

Suppose that a new variable appears in the equation to be compiled; then it will certainly appear in the linear form built at run time. So it may be chosen in step 1 at compile time. Much of the work in step 2 can also be compiled away. Step 3 is not needed because the variable is new. Only step 4 must be executed at runtime, so a new instruction is needed, which executes it and always succeeds. Similar simplifications can be made for the compilation of inequalities which contain a new variable. The presence of a new unconstrained variable in an inequality can in fact significantly simplify the work made by the inequality solver. The new instructions are:

$$solve\_no\_fail\_eq(l), solve\_no\_fail\_ge(l), solve\_no\_fail\_gt(l)$$

The instruction *solve_no_fail_eq* contains a special call to the equality solver containing the terms to be equated and the variable that shall become nonparametric:

```
if $code(p) = solve\_no\_fail\_eq(l)$
then $succeed$
     if $add\_counter = 0$
     then $l \leftarrow \langle Numb, accval(accbottom)\rangle$
     else $mk\_s\_unfree(l)$
          $equate\_no\_fail(svtop, accbottom)$
```

where

```
$mk\_s\_unfree(l)$  $\equiv$  $l \leftarrow \langle S\_var, svtop\rangle$
                      $svar\_lpf(svtop) := pftop$
                      $svar\_occ(svtop) := []$
                      $svar\_value(svtop) := nil$
                      $svtop := svtop+$
```

The update *equate_no_fail*(l, a) has the same effect as a call to the solver in order to equate the identifier l, chosen to become nonparametric, with the linear form built in the accumulator.

The instructions for inequalities are quite similar:

```
if $code(p) = solve\_no\_fail\_ge(l)$  |  $solve\_no\_fail\_gt(l)$
then $mk\_s\_unbound(l)$
     $store\_v(\geq, svtop)$  |  $store\_v(>, svtop)$
     $top_{\geq} := pftop$  |  $top_{>} := pftop$
     $inequality\_no\_fail(pftop, svtop)$
     $succeed$
```

where

```
$store\_v(type, l)$  $\equiv$  $pfval(pftop) := type$
                      $pfval(pftop+) := active$
                      $pfval(pftop + 2) := add\_counter$
                      $pfval(pftop + 3) := \langle accval(accbottom), nil\rangle$
                      **seq** $j = 1, \ldots, add\_counter$
                        $pfval(pftop + 3 + j) := accval(accbottom + j)$
                      **endseq**
                      $pfval(pftop + add\_counter + 4) := \langle 1, l\rangle$
                      $pftop := pftop + add\_counter + 5$
```

$inequality\_no\_fail(l_{pf}, l_{id})$ has the same effect as $inequality(l_{pf})$, except that the inequality solver can exploit the presence of the new identifier l_{id}.

Since *put_lpf* is used only for the first occurrence of a variable, a similar optimization can be made for the instruction $put\_lpf(x_i)$, which assigns x_i to a new identifier. This comes up to replace in the *put_lpf*-rule

$mk\_s\_unbound(x_i)$	by	$mk\_s\_unfree(x_i)$
$trigger\_equate(x_i, accbottom)$		$equate\_no\_fail(svtop, accbottom)$

By this transformation rules $put\_lpf(x_i)$ and $solve\_no\_fail\_eq(x_i)$ become identical, so that *put_lpf* can be eliminated.

When building a linear form in the accumulator, the cases where the constant is zero or the coefficient is 1 or -1 occur in the majority of instances. Special instructions can be introduced to cater for these commonly occurring cases in order to keep down the code size and cut down on decode time. The new instructions are:

- *initpf_0*
  start a new linear form with constant 0;
- $addpf\_va\{lr\}\_\{+-\}$[14](x_i)
  add a term consisting of a temporary variable with coefficient 1 or -1;
- $addpf\_va\{lr\}\_\{+-\}(y_n)$
  add a term consisting of a permanent variable with coefficient 1 or -1.

We omit to spell out the simple rules for these new instructions.

Note also that $get\_lpf(x_i)$ can be replaced by

$$addpf\_val\_-(x_i),\ solve\_eq0$$

[14]The brace notation stands for the instructions *addpf_var_+*, *addpf_var_-*, *addpf_val_+* and *addpf_val_-*.

The instruction *get_lpf* may then be entirely dispensed with. Through elimination of *set_svariable*, *put_lpf* and *get_lpf* we obtain the same instruction set as described in [10].

Bibliography

1. H.Aït-Kaci, *Warren's Abstract Machine. A tutorial reconstruction*, MIT Press, 1991.
2. C.Beierle & E.Börger, *Correctness proof for the WAM with types.* In: *Computer Science Logic* (Eds. E. Börger, G. Jäger, H. Kleine Büning, M. Richter), Springer LNCS 626, 1992, 15–34.
3. E.Börger & D.Rosenzweig, *An analysis of Prolog Database Views and their Uniform Implementation*, In: *Prolog. Paris Papers-2.* ISO/IEC JTC1 SC22 WG17 Prolog Standardization Report no. 80, July 1991, pp. 87–130.
   = CSE-TR-89–91, University of Michigan, Ann Arbor, Michigan 1991.
4. E.Börger & D.Rosenzweig, *The WAM—Definition and Compiler Correctness*, to appear in Logic Programming: Formal Methods and Practical Applications (Eds. C.Beierle, L.Plümer), Studies in Computer Science and Artificial Intelligence, North-Holland, 1994.
5. E.Börger & D.Rosenzweig, *Mathematical Definition of Full Prolog*, to appear in Science of Computer Programming, 1994.
6. Y.Gurevich, *Evolving Algebras. A Tutorial Introduction* in: Bulletin of the European Association for Theoretical Computer Science, no.43, February 1991, pp. 264–284.
7. Y.Gurevich, *Logic and the Challenge of Computer Science*, in: E.Börger (Ed.), Trends in Theoretical Computer Science. Computer Science Press, Rockville MA 1988, pp. 1–57.
8. J.Jaffar, S.Michaylov, P.J.Stuckey & R.H.C.Yap, *The CLP($\mathcal{R}$) Language and System*, ACM Transactions on Programming Languages and Systems, July 1992, pp 339–395.
9. Nevin C.Heintze, Joxan Jaffar, Spiro Michaylov, Peter J.Stuckey & Roland H.C.Yap, *The CLP($\mathcal{R}$) Programming Manual, Version 1.2*, September 1992.
10. J.Jaffar, S.Michaylov, P.J.Stuckey & R.H.C.Yap, *An Abstract Machine for CLP($\mathcal{R}$)*, November 1992.
11. R.F.Salamone, *An Abstract Modular Specification of the CLAM*, Tesi di Laurea, Dipartimento di Informatica, Università di Pisa, July 1993.

Appeared in: E.Börger (Ed.), *Specification and Validation Methods*, Oxford University Press, 1995, pp.IX+460.