[**To appear in:** B. Pehrson, I. Simon (eds): *Proceedings of the IFIP Congress 94*, Elsevier, Amsterdam, 1994]

Formal Design of an Abstract Machine for Constraint Logic Programming

Christoph Beierle[a]

[a]Fachbereich Informatik, Fernuniversität Hagen, D-58084 Hagen, Germany
e-mail: *Christoph.Beierle@fernuni-hagen.de*

Abstract: By studying properties of CLP over an unspecified constraint domain $\mathcal{X}$ one obtains general results applicable to all instances of CLP($\mathcal{X}$). The purpose of this paper is to study a *general implementation scheme* for CLP($\mathcal{X}$) by designing a generic extension WAM($\mathcal{X}$) of the WAM and a corresponding generic compilation scheme of CLP($\mathcal{X}$) programs to WAM($\mathcal{X}$) code which is based on Börger and Rosenzweig's WAM specification and correctness proof. Thus, using the evolving algebra specification method, we obtain not only a formal description of our WAM($\mathcal{X}$) scheme, but also a mathematical correctness proof for the design.*

Keyword Codes: D.1.6, F.3.1
Keywords: Logic Programming, Specifying and Verifying about Programs

1. INTRODUCTION

Recently, Gurevich's evolving algebra approach has not only been used for the description of the (operational) semantics of various programming languages (Modula-2, Occam, Prolog, Prolog III, Smalltalk, Parlog, C; see [9]), but also for the description and analysis of implementation methods: Börger and Rosenzweig [6] provide a mathematical elaboration of Warren's Abstract Machine [14,1] for executing Prolog. The description consists of several refinement levels together with correctness proofs, and a correctness proof w.r.t. Börger's phenomenological Prolog description [5]. Based on Börger and Rosenzweig's WAM description, [2] provides a mathematical specification of the PAM, a WAM extension to type-constraint logic programming, and proves its correctness w.r.t. PROTOS-L, a logic programming language with polymorphic order-sorted types [4]. [7] extends the correctness proof in [6] to the compilation of CLP($\mathcal{R}$) programs to the Constraint Logic Arithmetic Machine (CLAM) [12], [11]. These approaches demonstrate how the evolving algebra approach naturally allows for modifications and extensions in the description of both the semantics of programming languages as well as in the description of implementation methods.

However, both [2] and [7] aim at the specification and verification of specific systems that are within the constraint logic programming paradigm. CLP($\mathcal{R}$) is a particular instance (by arithmetic constraints over the real numbers) of CLP [10], a framework for constraint handling in logic programming; similarly, the type constraints of PROTOS-L

*Acknowledgements: I am most grateful to Egon Börger for many fruitful and encouraging discussions; the research reported here has profited very much from our joint work described in [2] and [3].

could be seen as an instance CLP($\mathcal{TYPE}$). On the other hand, many other approaches to constraint logic programming can be seen as instances of CLP. E.g., Trilogy [13] could be viewed as CLP($\mathcal{Z}$), or Prolog III [8] could be viewed as instantiating CLP by constraints over boolean and rational terms. Thus, by studying properties of CLP over an unspecified constraint domain $\mathcal{X}$ (denoted by CLP($\mathcal{X}$) in the sequel) one obtains general results applicable to all instances of CLP($\mathcal{X}$). The purpose of this paper is to study a *general implementation scheme* for CLP($\mathcal{X}$) by designing a generic extension WAM($\mathcal{X}$) of the WAM and a corresponding generic compilation scheme of CLP($\mathcal{X}$) programs to WAM($\mathcal{X}$) code. Based on the mathematical WAM specification and correctness proof in [6], we extend the work of [2] to general constraints. Thus, we obtain not only a formal description of our WAM($\mathcal{X}$) scheme, but also a mathematical correctness proof for the design.

Although in this paper we keep the domain $\mathcal{X}$ of constraints abstract, we will provide one further step of reuse of logic programming and WAM technology (and of the evolving algebra specification of [6] as well). Since a constraint solver for $\mathcal{X}$ also gives us a constraint solver for equational constraints in $T_\Sigma(\mathcal{X})$, the algebra of freely generated terms over $\mathcal{X}$, we will consider such constraints over $T_\Sigma(\mathcal{X})$. The equational constraint solving part will be done by term unification properly extended to interface with the (non-equational) constraint solver over $\mathcal{X}$. An important aspect of this paper will thus be the exact specification of the interaction between the unification part (which will be refined down to the basic WAM level, representing one of the major WAM principles) and the - still abstract - general constraint solver. For notational convenience we will, however, stick to the names CLP($\mathcal{X}$) and WAM($\mathcal{X}$). Furthermore, we assume that the reader is familiar with the WAM and the WAM specification given in [6]. For further details of the WAM($\mathcal{X}$) specification we refer to [3].

2. AN ABSTRACT NOTION OF CONSTRAINTS

The basic universes and functions in CLP($\mathcal{X}$) algebras dealing with terms and substitutions are taken directly from the standard Prolog algebras. For a formalization of constraints - in the spirit of constraint logic programming - we introduce a new abstract universe *X-CONSTRAINT*, connected to terms via the function

$$\texttt{dom:}\quad \textit{X-CONSTRAINT} \rightarrow \textit{VARIABLE}^*$$

yielding all variables occurring in an *X-CONSTRAINT*, called the *domain* of a constraint. Constraints are then defined as equations or (X-)constraints, i.e.

$$\textit{CONSTRAINT} \subseteq \textit{EQUATION} \cup \textit{X-CONSTRAINT}$$

Let *CSS* denote the set of all sets of constraints together with $\texttt{nil} \in \textit{CSS}$ denoting an inconsistent constraint system. The unifiability notion of ordinary Prolog is now replaced by a more general (for the moment abstract) constraint solving function:

$$\texttt{solvable:}\quad \textit{CSS} \rightarrow \textit{BOOL}$$

telling us whether the given constraint system is solvable or not. From every (solution of a) solvable constraint system we can extract a substitution part. Thus, we introduce a function

$$\texttt{solution:}\quad \textit{CSS} \rightarrow \textit{SUBST} \times \textit{CSS} \cup \{\texttt{nil}\}$$

where `solution(CS) = nil` iff `solvable(CS) = false`. For the trivially solvable empty constraint system we have `solution(`$\emptyset$`) = (`$\emptyset$`,`$\emptyset$`)` and the functions

$\texttt{subst\_part:} \quad CSS \rightarrow SUBST$
$\texttt{cs\_part:} \quad CSS \rightarrow CSS$

are the two obvious projections of `solution`. Since we want to consider equational constraints over the term algebra $T_\Sigma(\mathcal{X})$ freely generated over $\mathcal{X}$, we have to express that in `solution(CS) = (s,c)` the two solution components `s` and `c` are compatible with this requirement. That is, either the domains of `s` and `c` are disjoint, or, if a variable `x` is in both domains, in the substitution part it can only be identified with another variable since as a constraint variable it must still be a free variable. Thus we require:

if $\texttt{solution(CS) = (s,c)} \neq \texttt{nil}$ **then**
$\forall\ \texttt{x} \in \texttt{dom(s)} \cap \texttt{dom(c)}\ .\ \ \exists\ \texttt{y} \in \mathit{VARIABLE}\ .$
$\texttt{x} \doteq \texttt{y} \in \texttt{s}$ and, if $\texttt{y} \in \texttt{dom(c)}$, then $\texttt{x = y}$ in any interpretation of `c`.

Having refined the notions of unifiability and substitution to constraint solvability and (solvable) constraint system, respectively, we can now also refine the related notion of substitution result to terms with constrained variables. The latter involves three arguments:

1. a term `t` to be instantiated,
2. constraints for the variables of `t` (and possibly other variables) given by $\texttt{c}_\texttt{t}$, and
3. a constraint system `CS` (containing in general also a substitution part) to be applied.

Since a `CS`-solution consists of an ordinary substitution $\texttt{s}_\texttt{CS}$ together with constraints $\texttt{c}_\texttt{CS}$ via $\texttt{solution(CS) = (s}_\texttt{CS}\texttt{, c}_\texttt{CS}\texttt{)}$, the result of the constraint application can be introduced by

$$\texttt{conres(t, c}_\texttt{t}\texttt{, CS) = (t}_1\texttt{, c}_1\texttt{)}$$

as a pair consisting of the instantiated term $\texttt{t}_1$ and constraints $\texttt{c}_1$. For this function

$$\texttt{conres:} \quad \mathit{TERM} \times \mathit{CSS} \times \mathit{CSS} \rightarrow \mathit{TERM} \times \mathit{CSS} \cup \{\texttt{nil}\}$$

we impose the following integrity constraints:

$\forall\ \texttt{t} \in \mathit{TERM},\ \texttt{c}_\texttt{t} \in \mathit{CSS},\ \texttt{CS} \in \mathit{CSS}\ .$
if $\texttt{solvable(c}_\texttt{t} \cup \texttt{CS)}$ **then**
$\texttt{conres(t, c}_\texttt{t}\texttt{, CS) = (t}_1\texttt{, c}_1\texttt{)}$
where:
$\texttt{t}_1 = \texttt{subres(t,subst\_part(c}_\texttt{t} \cup \texttt{CS))}$
$\texttt{c}_1 = \texttt{cs\_part(c}_\texttt{t} \cup \texttt{CS)}$
else
$\texttt{conres(t, c}_\texttt{t}\texttt{, CS) = nil}$

Thus, the condition that a constraint system `CS` "can be applied" to a term `t` with variables constrained by $\texttt{c}_\texttt{t}$ means that $\texttt{c}_\texttt{t}$ is compatible with `CS`, i.e. $\texttt{solvable(CS} \cup \texttt{c}_\texttt{t}\texttt{) = true}$.

3. CLP($\mathcal{X}$) ALGEBRAS WITH COMPILED AND / OR STRUCTURE

As our starting point we can take CLP($\mathcal{X}$) algebras with compiled AND/OR structure. This is motivated by the fact that the constraint mechanism is orthogonal both to the compilation of the predicate structure (OR structure) as well as to the compilation of the clause structure (AND structure). Leaving the notion of terms and substitutions as abstract as in 2, we can use the compiled AND/OR structure development for Prolog

in [6] which was also used for PROTOS-L in [2] and for CLP($\mathcal{R}$) in [7]. Essentially we just have have to replace substitutions (resp. substitutions with type constraints) by the more general constraint systems, where the new instruction `add_constraint` enlarges the current constraint system accumulated so far.

4. TERM REPRESENTATION

The representation of terms and substitutions in the WAM($\mathcal{X}$) can be introduced in several steps. Following the development in [6] we first introduce the treatment of the low-level run-time unification but we keep the (non-equational) constraints completely abstract. Thus, in the WAM($\mathcal{X}$) algebras to be developed here we introduce a 0-ary function `xcs` holding the current non-substitutional constraint system accumulated so far. The only deviation from the WAM's resp. PAM's term representation is that we want to distinguish between data area locations `l` representing free variables with no constraints at all (`tag(l) = VAR`) and free constrained variables (`tag(l) = CVAR`).

4.1. Unification

Low-level unification in the WAM(X) can be carried out as in the WAM (see [1]) if we refine the bind operation into one that takes into account also the constraints of the variables (see [4] for the case of type constraints). The bind operation may thus also fail and initiate backtracking if the constraints are not satisfied. Thus, we can use the treatment of unification as described in [6], while leaving the bind operation abstract for the moment, not only in order to postpone the discussion of occur check and trailing but also to stress the fact that the bind operation will take care of the constraints for the variables. For the abstract `bind` update we impose the following modified

BINDING CONDITION 1: For any $\mathtt{l}_1$, $\mathtt{l}_2$, $\mathtt{l} \in$ *DATAARRA* with $\mathtt{unbound(l}_1\mathtt{)}$, with `term`, `term'` values of `term(l)` before and after execution of $\mathtt{bind(l}_1\mathtt{,l}_2\mathtt{)}$, we have: If

$$\mathtt{xcs' = xcs} \cup \{\mathtt{mk\_var(l}_1\mathtt{)} \doteq \mathtt{term(l}_2\mathtt{)}\}$$

is solvable, then `term' = conres(term, xcs')` and `cs_part(xcs')` will be the new value of `xcs`; otherwise backtracking will be executed.

With this generalized binding assumption we easily obtain a correspondingly modified unification lemma, expressing the effect of the $\mathtt{unify(l}_1\mathtt{,l}_2\mathtt{)}$ update.

4.2. Getting of terms

Whereas the compilation of body goals and the required term creation by putting instructions does not involve unification and thus remains unchanged, the getting of terms does involve unification. Parts of it are compiled into the getting instructions (like `get_structure` followed by a sequence of `unify` instructions) and the remaining unification tasks are handled by the low-level `unify` procedure.

The `get_value`, `unify_value`, and `unify_variable` instructions are as in the WAM or the PAM case. The first `get_structure` rule for WAM($\mathcal{X}$) is still as before, covering the situation when in $\mathtt{get\_structure(f,x}_i\mathtt{)}$ $\mathtt{x}_i$ is bound to a non-variable term. When $\mathtt{x}_i$ is unbound, it must be bound to a newly created term with top-level symbol `f`. Whereas in the WAM this will always succeed, in the WAM($\mathcal{X}$) case a possible constraint of $\mathtt{x}_i$ must be taken into account:

```
if   RUN
  & code(p) = get_structure(f,x_i)
  & unbound(deref(x_i))
  & NOT(constrained(x_i))     | constrained(x_i)
then
  h ← <STRUC,h+>              | backtrack
  bind(deref(x_i),h)          |
  val(h+) := f                |
  h := h++                    |
  mode := Write               |
  succeed                     |
```

The Getting Lemma now takes into account the constraint system accumulated in `xcs`.

4.3. Putting of Constraints

The compile function is refined to issue a new `put_constraint` instruction (for realization of `add_constraint` of Section 3):

```
if   RUN & code(p) = put_constraint(C)
then addtoxcs(ρ(C))
```

where ρ is the variable renaming substitution $\{Y_i \doteq \texttt{mk\_var}(y_i) \mid i=1,\ldots,l\}$, `vars(C)` = $\{Y_1,\ldots,Y_l\}$, and where the crucial `addtoxcs` update must satisfy the **ADD CONSTRAINT CONDITION:** For any `l` $\in$ *DATAAREA* with `term, term'` values of `term(l)` before and after execution of `addtoxcs(C)` we have: If `xcs' = xcs` $\cup$ {`C`} is solvable then `term' = conres(term,xcs')` and `cs_part`(`xcs'`) will be the new value of `xcs`, and any of the variables in `vars(C)` will be tagged as a constraint variable (i.e. `constrained(deref(`y_i`))`); otherwise backtracking will be executed.

5. CLP($\mathcal{X}$) ALGEBRAS

5.1. Environment and Choicepoint Representation

The stack of states and environments of CLP($\mathcal{X}$) algebras remains unchanged except that we extend the choice point information by a location `xcs(l)` $\equiv$ `l - 6` to hold the current (non-equational part of the) constraint system. It is stored in the stack when a new choice point is created (by executing a `try_me_else` or `try` instruction) and retrieved when updating or removing a choice point (by a `retry[_me_else]` or `trust[_me_else]` instruction).

5.2. Trailing

Trailing is done exactly as in the PAM, i.e. compared to the WAM not only the location `l` but also its value `val(l)` is trailed and recovered upon backtracking, ensuring that `VAR` and `CVAR` tags are restored correctly. While still leaving the binding update abstract, we pose the following

TRAILING CONDITION: Let l_1, l_2, `l` $\in$ *DATAAREA*, `C` $\in$ *CSS*. If `val(l)` before execution of `bind(`l_1`,`l_2`)` (resp. `addtoxcs(C)`) is different from `val(l)` after successful execution of `bind(`l_1`,`l_2`)` (resp. `addtoxcs(C)`), then the location `l` has been trailed with `trail(l)`.

For the WAM($\mathcal{X}$) algebras developed so far from CLP($\mathcal{X}$) algebras with compiled AND/OR structure we can now generalize the "Pure Prolog Theorem" of [6].

5.3. Additional WAM optimizations in the WAM($\mathcal{X}$)

Environment trimming and last call optimization (LCO) are among the most prominent optimizations in the WAM; other crucial points in the WAM design is the classsification of variables into temporary and permanent variables, as well as their initialization "on the fly", or the indexing methods. For a discussion of these aspects we refer to [1] and [6]. In the case of type constraints we showed in [2] how the treamtment of these WAM optimizations in [6] carried over to the PAM, including the realization of the Cut operator. The same argumentation applies to the WAM($\mathcal{X}$) extension since no constraint related actions are involved, allowing us to extend the correctnes theorems of [6] to the full WAM($\mathcal{X}$) with all WAM characteristics like environment trimming, LCO, or indexing.

REFERENCES

1. H. Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction.* MIT Press, Cambridge, MA, 1991.
2. C. Beierle and E. Börger. Correctness proof for the WAM with types. In E. Börger, H. Kleine Büning, G. Jäger, and M. M. Richter, editors, *Computer Science Logic - CSL'91*, volume 626 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1992.
3. C. Beierle and E. Börger. Evolving algebra specification of an abstract machine for constraint logic programming. Technical Report, FernUniversität Hagen, 1994. (to appear).
4. C. Beierle and G. Meyer. Run-time type computations in the Warren Abstract Machine. *The Journal of Logic Programming*, 18(2):123–148, February 1994.
5. E. Börger. A logical operational semantics of full Prolog. Part I. Selection core and control. In E. Börger, H. Kleine Büning, and M. M. Richter, editors, *CSL'89 - 3rd Workshop on Computer Science Logic*, volume 440 of *Lecture Notes in Computer Science*, pages 36–64. Springer-Verlag, Berlin, 1990.
6. E. Börger and D. Rosenzweig. The WAM - definition and compiler correctness. In C. Beierle and L. Plümer, editors, *Logic Programming: Formal Methods and Practical Applications*, Studies in Computer Science and Artificial Intelligence. North-Holland, 1994. (to appear).
7. E. Börger and R. Salamone. CLAM specification for provably correct compilation of CLP(R) programs. In E. Börger, editor, *Specification and Validation Methods for Programming Languages and Systems*. Oxford University Press, 1994. (to appear).
8. A. Colmerauer. An introduction to Prolog-III. *Communications of the ACM*, 33(7):69–906, July 1990.
9. Y. Gurevich. Evolving algebras. A tutorial introduction. *EATCS Bulletin*, 43, February 1991.
10. J. Jaffar and J.-L. Lassez. Constraint logic programming. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages*, pages 111–119. ACM, January 1987.
11. J. Jaffar, S. Michaylov, P. Stuckey, and R. Yap. An abstract machine for CLP($\mathcal{R}$). Technical report, IBM Research Devision, 1992.
12. J. Jaffar, S. Michaylov, P. Stuckey, and R. Yap. The CLP($\mathcal{R}$) language and system. *ACM TOPLAS*, pages 339–395, July 1992.
13. P. Voda. The constraint language Trilogy. Technical report, Complete Logic Systems, North Vancouver, BC, Canada, 1988.
14. D. H. D. Warren. An Abstract PROLOG Instruction Set. Technical Report 309, SRI, 1983.