Formal Aspects of Computing (1995) 3: 1–000

Specification and correctness proof of a WAM extension with abstract type constraints*

Christoph Beierle[1] and Egon Börger[2]

[1] Fachbereich Informatik, FernUniversität Hagen, D-58084 Hagen, Germany; [2] Dipartimento di Informatica, Università di Pisa, Corso Italia 40, I-56100 Pisa, Italia

Keywords: Constraint logic programming, abstract machine, WAM, types, type constraint, correctness proof, evolving algebras

Abstract. We provide a mathematical specification of an extension of Warren's Abstract Machine for executing Prolog to type-constraint logic programming and prove its correctness. Our aim is to provide a full specification and correctness proof of a concrete system, the PROTOS Abstract Machine (PAM), an extension of the WAM by polymorphic order-sorted unification as required by the logic programming language PROTOS-L.
In this paper, while leaving the details of the PAM's type constraint representation and solving facilities to a sequel to this work, we keep the notion of types and dynamic type constraints abstract to allow applications to different constraint formalisms like Prolog III or CLP(R). This generality permits us to introduce modular extensions of Börger's and Rosenzweig's formal derivation of the WAM. Since the type constraint handling is orthogonal to the compilation of predicates and clauses, we start from type-constraint Prolog algebras with compiled AND/OR structure that are derived from Börger's and Rosenzweig's corresponding compiled standard Prolog algebras. The specification of the type-constraint WAM extension is then given by a sequence of evolving algebras, each representing a refinement level, and for each refinement step a correctness proof is given. Thus, we obtain the theorem that for every such abstract type-constraint logic programming system L, every compiler to the WAM extension with an abstract notion of types which satisfies the specified conditions, is correct.

Correspondence and offprint requests to: Christoph Beierle, Fachbereich Informatik, LG Praktische Informatik VIII, FernUniversität Hagen, Bahnhofstr. 48, D-58084 Hagen, Germany; e-mail: christoph.beierle@fernuni-hagen.de

* The first author was partially funded by the German Ministry for Research and Technology (BMFT) in the framework of the WISPRO Project (Grant 01 IW 206). He would also like to thank the Scientific Center of IBM Germany where the work reported here was started.

1. Introduction

Recently, Gurevich's notion of evolving algebra [Gur88] 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 [Gur91]), but also for the description and analysis of implementation methods: Börger and Rosenzweig ([BR91, BR92b, BR92a]) provide a mathematical elaboration of Warren's Abstract Machine ([War83], [AK91]) 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 [Bör90a, Bör90b].

In this work we 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. Based on Börger and Rosenzweig's WAM description we provide a mathematical specification of a WAM extension to type-constraint logic programming and prove its correctness. Note that thereby our treatment can be easily extended to cover also all extra-logical features (like the Prolog *cut*) whereas the WAM correctness proof of [Rus92] deals merely with SLD resolution for Horn clauses.

The extension of logic programming by types requires in general not only static type checking, but types are also present at run time (see e.g. [MO84], [GM86], [NM88], [Han88], [Han91], [Smo89]). For instance, if there are types and subtypes, restricting a variable to a subtype represents a constraint in the spirit of constraint logic programming. PROTOS-L ([Bei92], [BBM91], [Bei95]) is a logic programming language that has a polymorphic, order-sorted type concept (similar to the slightly more general type concept of TEL [Smo88]) and a complete abstract machine implementation, called PAM ([BMS91], [BM94]) that is an extension of the WAM by the required polymorphic order-sorted unification. Our aim is to provide a full specification and correctness proof of the concrete PAM system.

Here we keep the notion of types and dynamic type constraints sufficiently abstract to allow applications to different constraint formalisms. Since the type constraint handling is orthogonal to the compilation of predicates and clauses, we start from type-constraint Prolog algebras with compiled AND/OR structure that are derived from Börger's and Rosenzweig's corresponding compiled standard Prolog algebras. The specification of the type-constraint WAM extension is then given by a sequence of evolving algebras, each representing a refinement level. For each refinement step a correctness proof is given. As final result of this paper we obtain the theorem: For every such abstract type-constraint logic programming system L and for every compiler satisfying the specified conditions, compilation from L to the the WAM extension with an abstract notion of types is correct.

Although our description in this paper is oriented towards type constraints, it is modular in the sense that it can be extended to other constraint formalisms, like Prolog III [Col90] or CLP(R) [JL87], [JMSY90], as well. For instance, in [BS95] a specification of the CLAM, an abstract machine for CLP(R), is given along these lines, together with a correctness proof for CLP(R) compilation. [Bei94] extends the work reported here by studying a general implementation scheme for CLP(X) and designing a generic extension WAM(X) of the WAM. Nevertheless, in order to avoid proliferation of different classes of evolving al-

gebras, we will already speak here in terms of PROTOS-L and PAM algebras (instead of type-constraint Prolog and type-constraint WAM algebras).

In a sequel to this work ([BB96]) we will refine the type constraints to the polymorphic order-sorted types of PROTOS-L, again in several refinement steps. This allows us to develop a detailed and mathematically precise account of the PAM's compiled type constraint representation and solving facilities, and to prove its correctness w.r.t. PROTOS-L which we then obtain as the final correctness theorem [BB96].

This paper was written in 1992/93 and revises and extends our work presented in [BB91] and [BB92]. It is organized as follows: Section 2 introduces an abstract notion of (type) constraints and defines PROTOS-L algebras with compiled AND/OR structure, the starting point of our development. This already includes the treatment of indexing and switching instructions which on this level of abstraction carry over from the WAM to the PAM. Section 3 introduces the representation of terms. The stack representation of environments and choicepoints is given in Section 4 which also contains the "Pure PROTOS-L" theorem stating the correctness of the PAM algebras developed so far w.r.t. the PROTOS-L algebras of Section 2. Various WAM optimizations that are also present in the PAM (environment trimming, last call optimization, initialization "on the fly" of temporary and permanent variables) are described in Section 5. The notions of type constraint and constraint solving have been kept abstract through all refinement levels so far; thus, the whole development carried out applies to any type system satisfying the given abstract conditions.

Notation and prerequisites

In this section we first list those definitions which are necessary to the reader who is interested only in analysis of the PAM, reading our rules as 'pseudocode over abstract data', and not in checking the correctness proof (for which we rely more explicitely on the underlying methodology of evolving algebras; for background and a definition of this notion which is due to Y. Gurevich see [Gur91]).

The abstract data comes as elements of (not further analysed) sets (domains, *universes*). The operations allowed on universes will be represented by partial *functions*.

We shall allow the setup to *evolve* in time, by executing *function updates* of the form

`f(`t_1`,...,`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.

We shall also allow some of the universes (typically initially empty) to *grow* in time, by executing updates of form

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

where *updates* may (and should) depend on the t_i's, setting the values of some functions on *newly created* elements t_i of `A`.

The precise way our 'abstract machines' may evolve in time will be determined by a finite set of *rules* of the form

if `condition`
then `updates`

where *condition* or guard is a boolean, the truth of which triggers *simultaneous* execution of all updates listed in *updates*. Simultaneous execution helps us avoid coding to, say, interchange two values.

If at every moment at most one rule is applicable (which will in this paper always be the case), we shall talk about *determinism* - otherwise we might think of a daemon freely choosing the rule to fire. The forms obviously reducible to the above basic syntax, which we shall freely use as abbreviations, are **let** and **if then else**. The transition rule notation

if condition$_1$ | ... | condition$_n$
then updates$_1$ | ... | updates$_n$

with pairwise incompatible conditions condition$_i$ stands for the obvious set of n transition rules

if condition$_1$
then updates$_1$

if condition$_2$
then updates$_2$

...

if condition$_n$
then updates$_n$

We will also use the |-notation to separate alternative parts within more complex rule conditions and the corresponding update parts. For instance, the rule notation

```
if OK
  & code(p) = call(BIP)
  & BIP =
     true | fail      | cut
then
  succeed | backtrack | b := ct'(e)
          |           | succeed
```

deals with the built-in predicates `true`, `fail`, and `cut` and stands for the three rules

```
if OK
  & code(p) = call(BIP)
  & BIP = true
then
  succeed
```

```
if OK
  & code(p) = call(BIP)
  & BIP = fail
then
   backtrack
```

```
if OK
  & code(p) = call(BIP)
  & BIP = cut
then
  b := ct'(e)
  succeed
```

Also, we will often introduce abbreviations of the form `a` ≡ `term`. For instance, in the rules just given we used the three abbreviations

```
succeed ≡ p := p + 1

OK      ≡ stop = 0
```

```
backtrack ≡ if b = nil
            then stop := -1
            else p := p(b)
```

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.

Here are some more remarks on the formal background for the reader who is interested to follow our proofs.

Definition. An *evolving algebra* is a pair $(\mathtt{A}, \mathtt{R})$ where $\mathtt{A}$ is a first-order heterogeneous algebra with partial functions and possibly empty domains, and $\mathtt{R}$ is a finite system of *transition rules*. The transition rules are of form

if *condition* **then** *updates*

where *condition* is a boolean expression of the signature of $\mathtt{A}$ and *updates* is a finite sequence of updates of one of the following three forms:

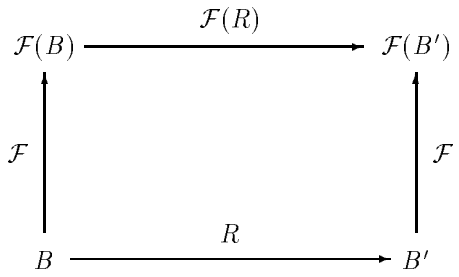
function update : $f(t_1, \ldots, t_n) := t$
where f is a function of $\mathtt{A}$ and $t_1, \ldots, t_n, t$ are terms in the signature of $\mathtt{A}$.

universe extension : **extend** A **by** $t_1, \ldots, t_n$ **with** *updates* **endextend**
where $t_1, \ldots, t_n$ are variables possibly occurring in function updates *updates* (standing for elements of A).

update schema : **FORALL** $i = t_1, \ldots, t_2$ **DO** *updates*(i) **ENDFORALL**
where t_1 and t_2 are numerical terms and *updates*(i) are updates (with parameter i).

The meaning of rules and updates execution is as explained above. We intend an update schema to denote an algebra update obtained by first evaluating t_1 and t_2 to numbers n_1 and n_2 and then executing *updates(i)* for all $i \in \{n_1, \ldots, n_2\}$ in parallel. This construct, which does not appear in Gurevich's original definition in [Gur91] is obviously reducible to rules with function updates.

Every evolving algebra $(\mathtt{A}, \mathtt{R})$ determines a class of structures called *algebras* or *states* of $(\mathtt{A}, \mathtt{R})$. Within such classes we will have a notion of *initial* and *terminal* algebras, expressing initial resp. final states of the target system. We are essentially interested only in those states which are reachable from inital states by $\mathtt{R}$. In our refinement steps we typically construct a more concrete evolving algebra $(\mathtt{B}, \mathtt{S})$ out of a given more abstract evolving algebra $(\mathtt{A}, \mathtt{R})$ and relate them by a (partial) *proof map* $\mathcal{F}$ mapping states B of $(\mathtt{B}, \mathtt{S})$ to states $\mathcal{F}(B)$ of $(\mathtt{A}, \mathtt{R})$, and rule sequences R of $\mathtt{R}$ to rule sequences $\mathcal{F}(R)$ of $\mathtt{S}$, so that the following diagram commutes:



In accordance to terminology used in abstract data type theory [EM89] we call $\mathcal{F}$ also an *abstraction function*.

We shall consider such a proof map to establish *correctness* of $(\mathtt{B},\mathtt{S})$ with respect to $(\mathtt{A},\mathtt{R})$ if $\mathcal{F}$ preserves initiality, success and failure (indicated by the value of a special 0-ary function `stop`) of states, since in that case we may view successful (failing) concrete computations as implementing successful (failing) abstract computations.

We can consider such a proof map to establish *completeness* of $(\mathtt{B},\mathtt{S})$ with respect to $(\mathtt{A},\mathtt{R})$ if every terminating computation in $(\mathtt{A},\mathtt{R})$ is image under $\mathcal{F}$ of a terminating computation in $(\mathtt{B},\mathtt{S})$, since in that case we may view every successful (failing) abstract computation as implemented by a successful (failing) concrete computation.

In case we establish, in the above sense, both correctness (as we will do explicitly on every of our refinement steps) as well as completeness (which follows from all our refinement steps by straightforward observations) we may speak of *operational equivalence* of evolving algebras.

2. PROTOS-L Algebras with compiled AND / OR structure

2.1. An abstract notion of type constraints

The basic universes and functions in PROTOS-L algebras dealing with terms and substitutions can be taken directly from the standard Prolog algebras ([Bör90a], [Bör90b]). In particular, we have the universes **TERM** and **SUBST** of terms and substitutions with a function

`subres`: **TERM** $\times$ **SUBST** $\rightarrow$ **TERM**

yielding `subres(t,s)`, the result of applying `s` to `t`.

To be able to talk about (type constraints of) variables involved in substitutions we introduce a new universe

VARIABLE $\subseteq$ **TERM**

Since in PROTOS-L unification on terms is subject to type constraints on the involved variables, we have to distinguish between equating terms and satisfying type constraints for them. For this purpose we introduce a universe

EQUATION $\subseteq$ **TERM** $\times$ **TERM**

whose elements are written as $\mathtt{t}_1 \doteq \mathtt{t}_2$. Substitutions are then supposed to be (represented by) finite sets of equations of the form $\{\mathtt{X}_1 \doteq \mathtt{t}_1, \ldots, \mathtt{X}_n \doteq \mathtt{t}_n\}$ with pairwise distinct variables $\mathtt{X}_i$. The domain of such a substitution is the set of variables occurring on the left hand sides. (Note: If you want to have the logically correct notion of substitution - with occur check -, you should add the condition that no $\mathtt{X}_i$ occurs in any of the $\mathtt{t}_j$.)

For a formalization of type constraints for terms - in the spirit of constraint logic programming - we introduce a new abstract universe **TYPETERM**, disjoint from **TERM** and containing all typeterms, of which we only assume that it comes with a special constant `TOP` $\in$ **TYPETERM**. Type constraints are given by the universe

TYPECONS $\subseteq$ **TERM** $\times$ **TYPETERM**

whose elements are written as $\mathtt{t:tt}$. A set $\mathtt{P} \subseteq \mathbf{TYPECONS}$ is called a *prefix* if it contains only type constraints of the form $\mathtt{X:tt}$ where $\mathtt{X} \in \mathbf{VARIABLE}$ and at most one such pair for every variable is contained in $\mathtt{P}$. The *domain* of $\mathtt{P}$ is the set of all variables $\mathtt{X}$ such that $\mathtt{X:tt}$ is in $\mathtt{P}$ for some $\mathtt{tt}$. We denote by **TYPEPREFIX** the universe of all type prefixes.

Constraints are then defined as equations or type constraints, i.e.

$$\mathbf{CONSTRAINT} \subseteq \mathbf{EQUATION} \cup \mathbf{TYPECONS}$$

Let **CSS** denote the set of all sets of constraints together with $\mathtt{nil} \in \mathbf{CSS}$ 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:

$$\mathtt{solvable}\colon \mathbf{CSS} \to \mathbf{BOOL}$$

telling us whether the given constraint system is solvable or not. Every (solution of a) solvable constraint system can be represented by a pair consisting of a substitution and a type prefix. Thus, we introduce a function

$$\mathtt{solution}\colon \mathbf{CSS} \to \mathbf{SUBST} \times \mathbf{TYPEPREFIX} \cup \{\mathtt{nil}\}$$

where $\mathtt{solution(CS) = nil}$ iff $\mathtt{solvable(CS) = false}$. For the trivially solvable empty constraint system we have

$$\mathtt{solution}(\emptyset) = (\emptyset,\emptyset)$$

and the functions

$$\begin{aligned}&\mathtt{subst\_part}\colon \ \mathbf{CSS} \to \mathbf{SUBST}\\ &\mathtt{prefix\_part}\colon \mathbf{CSS} \to \mathbf{TYPEPREFIX}\end{aligned}$$

are the two obvious projections of $\mathtt{solution}$. As an integrity constraint we assume

$$\mathtt{solution}(\{\mathtt{t:TOP}\}) = (\emptyset,\emptyset)$$

i.e., $\mathtt{TOP}$ is used to represent a trivially solvable type constraint.

These are the only assumptions we make about the universe **TYPETERM** in this paper. (A special representation for it is introduced in [BB96].) Thus, the complete development carried out here applies to any concept of (type) constraints that exhibits the minimal requirements stated so far.

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 type constrained variables. The latter involves three arguments:

1. a term $\mathtt{t}$ to be instantiated,
2. type constraints for the variables of $\mathtt{t}$ given by a prefix $\mathtt{P_t}$, and
3. a constraint system $\mathtt{CS}$ to be applied.

Since a $\mathtt{CS}$-solution consists of an ordinary substitution $\mathtt{s_{CS}}$ together with variable type constraints $\mathtt{P_{CS}}$ via $\mathtt{solution(CS) = (s_{CS},\ P_{CS})}$, the result of the constraint application can be introduced by

$$\mathtt{conres(t,\ P_t,\ CS) = (t_1,\ P_1)}$$

as a pair consisting of the instantiated term $\mathtt{t_1}$ and type constraints $\mathtt{P_1}$ for the variables of $\mathtt{t_1}$. For this function

$$\begin{aligned}\mathtt{conres}\colon\ &\mathbf{TERM} \times \mathbf{TYPEPREFIX} \times \mathbf{CSS} \to\\ &\mathbf{TERM} \times \mathbf{TYPEPREFIX} \cup \{\mathtt{nil}\}\end{aligned}$$

we impose the following integrity constraints:

$\forall$ t $\in$ **TERM**, $P_t \in$ **TYPEPREFIX**, CS $\in$ **CSS** .
if solvable($P_t \cup$ CS) **then**
conres(t, P_t, CS) = (t_1, P_1)
where:
t_1 = subres(t, subst_part(CS))
P_1 = prefix_part($P_t \cup$ CS)$_{|var(t_1)}$
else
conres(t, P_t, CS) = nil

where $P'_{|var(t')}$ is obtained from P' by eliminating the type constraints for all variables not occurring in t'.

$P\backslash X$ will be an abbreviation for $P_{|domain(P)\backslash\{X\}}$, the prefix obtained from P by eliminating (if present) the constraint for X.

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

2.2. Compilation

As already mentioned, our starting point in this paper are PROTOS-L algebras with compiled AND/OR structure. This is motivated by the fact that the type 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 2.1, we can use the compiled AND/OR structure development for Prolog in [BR91], [BR92b] also for PROTOS-L: Essentially we just have to replace substitutions by the more general constraint systems, and have to take care of a clause constraint when resolving a goal.

In a PROTOS-L algebra a program is a pair consisting of a definition context and a sequence of clauses

PROGRAM $\subseteq$ **DEFCONTEXT** $\times$ **CLAUSE***

The definition context contains declarations of types, type constructors, etc. and will be refined in [BB96]. For prog = (defc,db) $\in$ **PROGRAM** we will write x $\in$ prog for both x $\in$ defc and x $\in$ db when it is clear from the context whether x is e.g. a type declaration or a list of clauses. A clause, depicted as

{P} H <-- G_1 & ... & G_n.

is an ordinary Prolog clause together with a set P of type constraints for (all and only) the variables occurring in the clause head and body. As in [BS91] we use three obvious projection functions

clhead: **CLAUSE** $\rightarrow$ **LIT**
clbody: **CLAUSE** $\rightarrow$ **LIT***
clconstraint: **CLAUSE** $\rightarrow$ **TYPEPREFIX**

where **LIT** is the universe of literals. Literals as used in ordinary logic programming are (non-negated) atomic first-order formulas. An element of the universe **GOAL** also comes with a type prefix and is written as

{P} G_1 & ... & G_n.

We assume a universe **INSTR** of instructions containing

$$\{\texttt{unify(H), add\_constraint(P), call(G), allocate, deallocate, proceed, true, fail, cut, try\_me\_else(N), try(L), retry\_me\_else(N), retry(L), trust\_me, trust(L), switch\_on\_term(i,Lv,Ls), switch\_on\_structure(i,T)} \mid \texttt{i} \in \mathbf{NAT}, \texttt{H, G} \in \mathbf{TERM}, \texttt{P} \in \mathbf{TYPEPREFIX}, \texttt{N, L, Lv, Ls} \in \mathbf{CODEAREA}, \texttt{T} \in (\mathbf{ATOM} \times \mathbf{NAT} \times \mathbf{CODEAREA})^*\}$$

Here, `add_constraint` is a new instruction not occurring in the WAM that adds a clause constraint to the current set of constraints accumulated so far. The universe **ATOM** contains the constant and function symbols; elements of **ATOM** are used in the `switch_on_structure` instruction in order to allow indexing over the top-level function symbol of an argument. Later on, further instructions will be added to **INSTR**.[1]

For the compilation of clauses we have a function

$$\texttt{compile: } \mathbf{CLAUSE} \to \mathbf{INSTR}^*$$

$$\texttt{compile(\{P\} H <-- } G_1 \texttt{ \& ... \& } G_n\texttt{)} = \texttt{[allocate, add\_constraint(P), unify(H), call(}G_1\texttt{), ...call(}G_n\texttt{), deallocate, proceed]}$$

Compiled programs are "stored" in a universe **CODEAREA** which comes with functions

$$\texttt{+,-: } \mathbf{CODEAREA} \to \mathbf{CODEAREA}$$
$$\texttt{code: } \mathbf{CODEAREA} \to \mathbf{INSTR}$$

where `+` and its inverse `-` yield a linear structure on **CODEAREA** and `code(l)` gives the instruction "stored" in `l`. The function

$$\texttt{unload: } \mathbf{CODEAREA} \to \mathbf{INSTR}^*$$

$$\texttt{unload(Ptr)} = \textbf{if } \texttt{code(Ptr) = proceed } \textbf{then } \texttt{[proceed] } \textbf{else } \texttt{[code(Ptr)|unload(Ptr+)]}$$

is an auxiliary function. We say that $\texttt{Ptr} \in \mathbf{CODEAREA}$ points to *code for a clause* `Cl` if

$$\texttt{unload(Ptr) = compile(Cl)}$$

The function

$$\texttt{procdef: } \mathbf{LIT} \times \mathbf{CSS} \times \mathbf{PROGRAM} \to \mathbf{CODEAREA}$$

yields a pointer $\texttt{Ptr} = \texttt{procdef(G,Cs,Prog)}$ that points to a chain `chain(Ptr)` of clauses containing all candidate clauses for resolving `G` in `Prog` under the constraint system `Cs`, i.e.:

$$\forall\ \texttt{Cl} \in \texttt{Prog}\ .\ ((\forall\ \texttt{P} \in \texttt{chain(procdef(G,Cs,Prog))}\ .\ \texttt{P does not point to code for Cl}) \Rightarrow \texttt{solvable(\{g} \doteq \texttt{rename(clhead(Cl),i)\}} \cup \texttt{Cs} \cup \texttt{rename(clconstraint(Cl),i))} = \texttt{false})$$

where $\texttt{i} \in \mathbf{NAT}$ is chosen such that `rename(GC,i)` renames all variables in a goal or constraint `GC` to new variables. For the auxiliary function `chain`

[1] Note that in this paper we do not consider a special representation for constants or lists. These are present in the PAM, and could be added to our formal treatment without difficulty. For instance, `switch_on_term` would get an additional argument for the constant case.

chain: **CODEAREA** → **CODEAREA***

we assume for an activator literal act

chain(Ptr) =

chain(Lv)	if code(Ptr) = switch_on_term(i,Lv,Ls) and is_var(X_i)
chain(Ls)	if code(Ptr) = switch_on_term(i,Lv,Ls) and is_struct(X_i)
chain(select(T,f,a))	if code(Ptr) = switch_on_structure(i,T) and f = functor(X_i) and a = arity(X_i)
chain_1(Ptr)	otherwise

chain_1(Ptr) =

flatten[chain_1(Ptr),chain_1(N)]	if code(Ptr) = try_me_else(N) or code(Ptr) = retry_me_else(N)
flatten[chain_1(C),chain_1(Ptr+)]	if code(Ptr) = try(C) or code(Ptr) = retry(C)
chain_1(Ptr+)	if code(Ptr) = trust_me
chain_1(C)	if code(Ptr) = trust(C)
[Ptr]	otherwise

where X_i = arg(act,i), functor, arity, and arg are the term analyzing functions, and is_var and is_struct are true for variables and compound terms, respectively. Furthermore, the switch_on_structure parameter T could be thought of as a hash table, with select(T,f,a) = pt if (f,a,pt) ∈ T.

2.3. Choicepoints and Environments

Executing AND/OR compiled PROTOS-L programs requires two stacks where w.r.t. the Prolog case we replace the substitution part by a constraint system. **STATE** is a universe to store the choicepoints and comes with functions

nil:	→ **STATE**	
cs:	**STATE** → **CSS**	accumulated constraint system
p:	**STATE** → **CODEAREA**	program pointer
cp:	**STATE** → **CODEAREA**	continuation pointer
e:	**STATE** → **ENV**	environment
b:	**STATE** → **STATE**	backtracking point
vi:	**STATE** → **NAT**	renaming index for variables
ct:	**STATE** → **STATE**	cut point

The universe **ENV** of environments comes with functions

nil:	→ **ENV**	
ce:	**ENV** → **ENV**	continuation environment
cp':	**ENV** → **CODEAREA**	continuation pointer
ct':	**ENV** → **STATE**	cut point
vi':	**ENV** → **NAT**	renaming index for variables

As in the WAM, **STATE** and **ENV** are embedded into a single **STACK**

STATE, ENV ⊆ **STACK**

-: **STACK** → **STACK**

with a common bottom element nil. tos(b,e) denotes the *top of the stack* which will always be the maximum of b and e.

2.4. Initial State

To hold the current status of the machine there are some 0-ary functions which correspond to their unary counterparts above. Given the PROTOS-L goal {P} G_1 & ... & G_n we have the following initial values:

cs	∈ **CSS**	cs = ∅
p	∈ **CODEAREA**	unload(p) = [add_constraint(P), call(G_1),...,call(G_n), proceed]
cp	∈ **CODEAREA**	cp = p++
e	∈ **ENV**	vi'(e)=0, ct'(e)=nil, ce(e)=nil
b	∈ **STATE**	b = nil
vi	∈ **NAT**	vi = 0
ct	∈ **STATE**	ct = nil

The literals of the initial PROTOS-L goal, as well as all intermediate goals that will be constructed during program executing, can be recovered via the continuation pointer. For code(cp-) = call(G) (which will always be the case as long as there is still something to do) we have in particular

act ≡ subres(rename(G,vi'(e)),subst_part(cs))

which is called the *current activator*.

The 0-ary function prog ∈ **PROGRAM** holds all declarations and clauses of the program (which in this paper will always be constant since we do not consider database operations like assert or retract). Finally, stop ∈ {-1,0,1} indicates whether the machine has stopped with failure, is still running, or has stopped with success.

2.5. Transition rules

The transition rules are as in the Prolog case with the substitution component being replaced by a constraint system, and with the following extension to the unify rule and the new add_constraint instruction:

unify

if OK
  & code(p) = unify(H)
then
  let cs1 = {act ≐ rename(H,vi)}
  if solvable(cs ∪ cs1)
  then cs := cs ∪ cs1
       vi := vi + 1
       succeed
  else backtrack

add_constraint

if OK
  & code(p) = add_constraint(P)
then
  let cs1 = rename(P,vi)
  if solvable(cs ∪ cs1)
  then cs := cs ∪ cs1
       succeed
  else backtrack

The condition OK is an abbreviation for stop = 0, i.e., the machine is operating in normal mode and no stop condition has been encountered. All abbreviations as well as the complete set of transition rules are given in Appendix A.

3. Term representation

The representation of terms and substitutions in the WAM can be introduced in several steps. Following the development in [BR92b] we first introduce the treatment of the low-level run-time unification (but leaving the details of type constraint solving as an abstract update to be refined later), followed by the term constructing and analyzing put and get instructions. In particular, the WAM-specific optimizations of environment trimming, last call optimization, or the initialization of temporary and permanent variables are postponed until we have established the correctness of the first refinement level with respect to the PROTOS-L algebras with compiled AND/OR structure in Section 2. The major derivation from the real PAM code in Sections 3 and 4 will be our simplifying assumption that all variables are permanent and are initialized on allocation to free unconstrained variables. Under this assumption the variables receive their initial type restrictions, derived statically by the compiler, immediately after allocation. This is achieved by a new (auxiliary) `put_constraint` instruction which will be dropped again later (in Section 5).

3.1. Universes and Functions

For the representation of terms we use the pointer algebra

(DATAAREA; `+, -; val`**)**

with **DATAAREA** $\subseteq$ **MEMORY**, where

`+, -` : **DATAAREA** $\rightarrow$ **DATAAREA**

connect the locations in **DATAAREA** and are inverse to each other. In the codomain of the function

`val`: **DATAAREA** $\rightarrow$ **PO + MEMORY + SYMBOLTABLE**

we use the universe **SYMBOLTABLE** in order to connect a function symbol to its arity and type. It comes with functions

`atom`: **SYMBOLTABLE** $\rightarrow$ **ATOM**
`arity`: **SYMBOLTABLE** $\rightarrow$ **NAT**
`entry`: **ATOM** $\times$ **NAT** $\rightarrow$ **SYMBOLTABLE**

of which we assume `atom(entry(f,n)) = f` and `arity(entry(f,n)) = n` for any atom `f` with arity `n`, and `entry(atom(s),arity(s)) = s` for any `s` $\in$ **SYMBOLTABLE**.

The functions `tag` and `ref` are defined on the universe **PO** of "PROTOS-L objects"

`tag`: **PO** $\rightarrow$ **TAGS**
`ref`: **PO** $\rightarrow$ **DATAAREA + TYPETERM**

where, because of the type constraint treatment, a new tag `VAR` for indicating free variables is introduced into the universe

TAGS = {`REF, STRUC, VAR`}

Special tags for representing constants, lists, built-in integers, etc. are also present in the PAM, but in this paper we consider them as optimizations that can be added later on without any difficulties. The tag `FUNC` from [BR92b] is not included since it is not needed.

The codomain of `ref` contains the universe **TYPETERM** since we will keep the type term representation abstract here; it will be refined later (see Section 2 in [BB96]).

As in [BR92b] we use some abbreviations for dealing with locations `l` $\in$ **DATAAREA**:

```
tag(l)          ≡ tag(val(l))
ref(l)          ≡ ref(val(l))
l1 ← l2         ≡ val(l1) := val(l2)
l ← <t,r>       ≡ tag(l) := t
                  ref(l) := r
unbound(l)      ≡ tag(l) = VAR
mk_unbound(l)       ≡ mk_unbound(l,TOP)
mk_unbound(l,tt)    ≡ tag(l) := VAR
                      insert_type(l,tt)
insert_type(l,tt) ≡ ref(l) := tt
```

where the last four abbreviations deal with the typed variable representation and where `tt` $\in$ **TYPETERM**. Note that an unconstrained free variable gets the trivial type restriction `TOP`, representing no restriction at all (c.f. Section 2.1).

In addition to the (partial) dereferencing and term reconstructing functions from the WAM case we now also assume a function that recovers the type constraints for all variables occurring in a term. Of these functions

```
deref:       DATAAREA → DATAAREA
term:        DATAAREA → TERM
type_prefix: DATAAREA → TYPEPREFIX
```

we assume for `l` $\in$ **DATAAREA**:

$$\texttt{deref(l)} = \begin{cases} \texttt{deref(ref(l))} & \text{if } \texttt{tag(l)} = \texttt{REF} \\ \texttt{l} & \text{otherwise} \end{cases}$$

$$\texttt{term(l)} = \begin{cases} \texttt{mk\_var(l)} & \text{if } \texttt{unbound(l)} \\ \texttt{term(deref(l))} & \text{if } \texttt{tag(l)} = \texttt{REF} \\ \texttt{f}(\texttt{a}_1,\ldots,\texttt{a}_n) & \text{if } \texttt{tag(l)} = \texttt{STRUC} \text{ and } \\ & \quad \texttt{f} = \texttt{atom(val(ref(l)))} \\ & \quad \texttt{n} = \texttt{arity(val(ref(l)))} \\ & \quad \texttt{a}_i = \texttt{term(ref(l)+i)} \end{cases}$$

$$\texttt{type\_prefix(l)} = \begin{cases} \texttt{mk\_var(l)}:\texttt{ref(l)} & \text{if } \texttt{unbound(l)} \\ \texttt{type\_prefix(deref(l))} & \text{if } \texttt{tag(l)} = \texttt{REF} \\ \texttt{P}_1 \cup \ldots \cup \texttt{P}_n & \text{if } \texttt{tag(l)} = \texttt{STRUC} \text{ and } \\ & \quad \texttt{n} = \texttt{arity(val(ref(l)))} \\ & \quad \texttt{P}_i = \texttt{type\_prefix(ref(l)+i)} \end{cases}$$

where `make_var(l)` $\in$ **VARIABLE** is a unique variable assigned to `l`. Note that the condition `term(l)` $\in$ **TERM** now implies various consistency properties like:

```
if unbound(l)                 then  ref(l) ∈ TYPETERM
if tag(l) ∈ {REF, STRUC}      then  ref(l) ∈ DATAAREA
                                    term(l) ∈ TERM
                                    typeprefix(l) ∈ TYPEPREFIX
if tag(l) = STRUC then  val(ref(l)) ∈ SYMBOLTABLE
```

term(ref(l)+i) ∈ **TERM**
type_prefix(ref(l)+i) ∈ **TYPEPREFIX**

with i ∈ {1,...,arity(val(ref(l)))}.

3.2. Unification

Lowlevel unification in the PAM can be carried out as in the WAM (see [AK91]) if we refine the bind operation into one that takes into account also the type constraints of the variables ([BMS91], [BM94]). The bind operation may thus also fail and initiate backtracking if the type constraints are not satisfied. Thus, we can use the treatment of unification as described in [BR92b], 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 type constraints for the variables.

To be more precise, the **DATAAREA** subalgebra

(PDL; pdl, nil; +, -; ref')

with pdl, nil ∈ **PDL** and

ref': **PDL**→ **DATAAREA**

is the *push down list* used for lowlevel unification, containing all pairs of (addresses of) terms still to be unified, with

left ≡ ref'(pdl) right ≡ ref'(pdl-)

being the current pair of terms. Unification is triggered by the update

```
unify(l1,l2) ≡ ref'(nil++) := l1
               ref'(nil+) := l2
               pdl := nil++
               what_to_do := Unify
```

The 0-ary function

what_to_do ∈ {Unify, Run}

will be used in the guard of all following rules in the form of conditions like

```
UNIF ≡ OK & what_to_do = Unify
RUN  ≡ OK & what_to_do = Run
```

Unification is carried out by unification rules as in [BR92b] (see appendix B.1) where for the abstract bind update we impose the following modified

BINDING CONDITION 1: For any l_1, l_2, l ∈ **DATAARRA**, with term resp. term' values of term(l) and with prefix resp. prefix' values of type_prefix(l) before resp. after execution of bind(l_1,l_2), we have if unbound(l_1) holds:

```
LET CS = {mk_var(l1) = term(l2)} ∪ type_prefix(l1)
                                 ∪ type_prefix(l2)
If solvable(CS) = true
then (term', prefix') = conres(term, prefix, CS)
else backtrack update will be executed.
```

With this generalized binding assumption we obtain the following modified

UNIFICATION LEMMA: If `pdl-- = nil`, `term(left)`, `term(right)` $\in$ **TERM**, and `type_prefix(left)`, `type_prefix(right)` $\in$ **TYPEPREFIX**, the effect of setting `what_to_do` to `Unify`, for any `l` $\in$ **DATAAREA** such that `term(l)` $\in$ **TERM** and `type_prefix(l)` $\in$ **TYPEPREFIX** is as follows:

Let `term` resp. `term'` be the values of `term(l)` and `prefix` resp. `prefix'` be the values of `type_prefix(l)` when setting `what_to_do` to `Unify` and when `what_to_do` has been set back to `Run` again, respectively. Then we have:

LET `CS = {term(left) = term(right)}` $\cup$ `type_prefix(left)` $\cup$ `type_prefix(right)`
If `solvable(CS) = true`
then `(term', prefix') = conres(term, prefix, CS)`
else `backtrack` update will be executed.

Proof. The proof of the Unification Lemma is by induction on the size of the terms to be unified, relying on our generalized Binding Condition. □

3.3. Putting of terms

As in the WAM, run time structures are created in the subalgebra of **DATAAREA**

(HEAP; `h, boh; +, -;` `val)`

where `h`, `boh` $\in$ **HEAP** represent the top resp. the bottom element of the heap. We use `nextarg` $\in$ **HEAP** to point to the next argument when anyalyzing a structure on the heap. Furthermore, we now assume

DATAAREA + CODEAREA $\subseteq$ **MEMORY**

where **CODEAREA** is as in Section 2.2 but where **INSTR** now contains

`put_value(`y_n`,`x_j`)`, `put_structure(f,`x_i`)`, `get_value(`y_n`,`x_j`)`,
`get_structure(f,`x_i`)`, `unify_value(`y_n`)`, `unify_variable(`x_n`)`,
`put_constraint(`y_n`,tt)`

with `n,j,i` $\in$ **NAT**, `f` $\in$ **SYMBOLTABLE**, `tt` $\in$ **TYPETERM**, $y_n \in$ **DATAAREA**, $x_i \equiv$ `x(i)`, where `x:` **NAT** $\rightarrow$ **AREGS** and **AREGS** $\subseteq$ **DATAAREA**. Note that `put_constraint(`y_n`,tt)` is a new instruction used for inserting a type restriction into a heap location. Instead of having a pair `(fn,a)` $\in$ **ATOM** $\times$ **NAT** we use `f = entry(fn,a)` in the code.

The code developed in Section 1.2 of [BR92b] for constructing terms in body goals uses put instructions which assume that, for all variables Y_i of the term `t` to be built on the heap, there is already a term denoting $y_i \in$ **DATAAREA** available. Since this means in particular that no variables are created during this process, we can use (with the obvious modification mentioned above) the same put instructions (i.e. `put_value`, `unify_value` in `Write` mode, `put_structure`) for the compilation of a body goal (see Appendix B.2 and B.3). Furthermore, we may assume that for the variables Y_i we have no type constraints to formalize here because they have already been associated to the corresponding location y_i (i.e. the variable `term(`y_i`)` which is - up to renaming - equal to Y_i. This gives us the following

PUTTING LEMMA: If all variables occurring in a term `t` $\in$ **TERM** are among $\{Y_1,\ldots,Y_l\}$, and if for `n` $\in$ {`1, ..., l`}, $y_n \in$ **DATAAREA** with

`term(`y_n`)` $\in$ **TERM**
`type_prefix(`y_n`)` $\in$ **TYPEPREFIX**

and X_i is a fresh variable, and `CS` is the constraint system consisting of the substitution associating every Y_n with `term(`y_n`)` and of the union of the type constraints `type_prefix(`y_n`)`, i.e.

`CS` = $\bigcup_n$ $\{Y_n \doteq$ `term(`y_n`)`$\}$ $\cup$ `type_prefix(`y_n`)`

then the effect of setting `p` to

`load(append(put_code(`X_i `= t), More))`

with subsequent fresh indices generated by the term normal form function nf_s (Appendix B.2) being non-top level, is that the pair

`(term(`x_i`), type_prefix(`x_i`))`

at the moment of passing to `More`, gets value of

`conres(t,` $\emptyset$`, CS)`

Proof. The proof follows by induction over the size of the involved terms, observing that no type related actions like variable creation or variable binding is involved here. □

3.4. Getting of terms

Unlike putting of terms that does not involve unification, the getting of terms does involve unification where 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 lowlevel `unify` procedure.

The `get_value`, `unify_value`, and `unify_variable` instructions are as in the WAM case (see Appendix B.4 and B.5). Note that we need `unify_variable` both in `Read` and `Write` mode which is controlled by the 0-ary function `mode` $\in$ {`Read, Write`}. In [BR92b] `unify_variable` in `Write` mode is introduced only as an optimization for variable initialization "on the fly", but when the machine enters `Write` mode in `get_structure`, `unify_variable` will be executed for the auxiliary substructure descriptors X_i generated by the term normal form function nf_a (Appendix B.2). Since we do not have to consider type contraints for such X_i, it suffices to initialize them to a free variable without any type restriction. Thus, for the generation of a heap variable in `Write` mode of `unify_variable` we use

```
mk_heap_var(l) ≡ mk_unbound(h)
                 bind(l,h)
                 h := h+
```

When `unify_variable` will be used for "on the fly" initialization of typed variables, we will have to consider an additional type initialization parameter (c.f. Section 5).

The first `get_structure` rule for PROTOS-L is as in the WAM case, covering the situation where `x`$_i$ in `get_structure(f,x`$_i$`)` is bound to a non-variable term (Appendix B.4). When `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

PAM case the type constraint of $\mathtt{x}_i$ must be taken into account. Indeed, what is happening here is the `binding` of a variable `X` with a type constraint, say `tt`, to a term `t` starting with `f`. In abstract terms this amounts to solving the constraint system

$$\{\mathtt{X} \doteq \mathtt{t},\ \mathtt{X}:\mathtt{tt}\}$$

We still want to leave the details of variable binding abstract here; what is of interest for this special case occurring in `get_structure` is which type constraints stemming from `tt` and (the declaration of) `f` must be propagated onto the argument terms of `t` = `f(...)`. Therefore, we introduce the function

$$\mathtt{propagate\_list}\colon \mathbf{SYMBOLTABLE} \times \mathbf{TYPETERM} \to \mathbf{TYPETERM}^* \cup \{\mathtt{nil}\}$$

yielding for arguments `entry(f,n)` and `tt` the list of type terms the arguments of `f` must satisfy. To be more precise, we have the following integrity constraint:

$$\mathtt{propagate\_list(entry(f,n),tt)} = (\mathtt{tt}_1,\ldots,\mathtt{tt}_n) \text{ iff } \mathtt{prefix\text{-}part}(\{\mathtt{f}(\mathtt{X}_1,\ldots,\mathtt{X}_n):\mathtt{tt}\}) = \{\mathtt{X}_{i_1}:\mathtt{tt}_{i_1},\ldots,\ \mathtt{X}_{i_k}:\mathtt{tt}_{i_k}\}$$

where $\{i_1,\ldots,i_k\} \subseteq \{1,\ldots,n\}$, and for $j \in \{1,\ldots,n\}\backslash\{i_1,\ldots,i_k\}$ we have $\mathtt{tt}_j$ = `TOP`.

If the constraint system $\{\mathtt{f}(\mathtt{X}_1,\ldots,\mathtt{X}_n):\mathtt{tt}\}$ is not solvable, no propagation is possible, and if it reduces to the trivially solvable empty constraint system, `propagate_list` yields a list containing only `TOP`. Thus we introduce the abbreviations

$$\mathtt{can\_propagate(entry(f,n),tt)} \equiv \mathtt{solution}(\{\mathtt{f}(\mathtt{X}_1,\ldots,\mathtt{X}_n):\mathtt{tt}\}) \neq \mathtt{nil}$$
$$\mathtt{trivially\_propagates(entry(f,n),tt)} \equiv \mathtt{solution}(\{\mathtt{f}(\mathtt{X}_1,\ldots,\mathtt{X}_n):\mathtt{tt}\}) = \emptyset$$

Get-Structure-2

```
if   RUN
  & code(p) = get_structure(f,x_i)
  & unbound(deref(x_i))
  & can_propagate(f,ref(deref(x_i)))
              = true                                  | = false
  & trivially_propagates(f,ref(deref(x_i)))          |
      = true    | = false                             |
then
  h ← <STRUC,h+>                                      | backtrack
  bind(deref(x_i),h)                                  |
  val(h+) := f                                        |
  h := h++                                            |
  mode := Write | nextarg := h++                       |
                | mk_unbounds(h+,propagate_list(f,ref(deref(x_i))) |
                | mode := Read                        |
  succeed                                             |
```

For $\mathtt{l} \in \mathbf{DATAAREA}$ and $\mathtt{tt}_1,\ldots,\mathtt{t}_n \in \mathbf{TYPETERM}$, the update

$$\mathtt{mk\_unbounds(l,(tt}_1,\ldots,\mathtt{tt}_n)) \equiv \mathtt{FORALL\ i = 1,\ldots,n\ DO\ mk\_unbound(l+i,tt}_i) \mathtt{\ ENDFORALL}$$

puts `n` type restricted variables at the locations `l+1,...,l+n` on the heap. When this update is executed in the rule above the machine continues in `read` mode so that the subsequent `n` unify instructions take into account these type restrictions.

GETTING LEMMA: If all variables occurring in a term `t` $\in$ **TERM** are among $\{Y_1, \ldots, Y_l\}$, and if for `n` $\in \{1, \ldots, l\}$, $y_n \in$ **DATAAREA** with

`unbound(`y_n`)`
`ref(`y_n`)` $\in$ **TYPETERM**

and X_i is a fresh variable with $x_i \in$ **DATAAREA** and

`term(`x_i`)` $\in$ **TERM**
`type_prefix(`x_i`)` $\in$ **TYPEREFIX**

and `CS` is the constraint system consisting of the equation `t` $\doteq$ `term(`x_i`)` together with `type_prefix(`x_i`)` and the union of the type constraints `type_prefix(`y_n`)`, i.e.

`CS = {t` $\doteq$ `term(`x_i`)} ` $\cup$ `type_prefix(`x_i`)` $\cup \bigcup_n$ `type_prefix(`y_n`)`

then the effect of setting `p` to

`load(append(get_code(`X_i `= t), More))`

for any `l` $\in$**DATAAREA** with `term = term(l)` $\in$ **TERM** and `typeprefix = type_prefix(l)` $\in$ **TYPEPREFIX** being the values before execution, is as follows:

If `solvable(CS)` = `true` then `p` reaches `More` without backtracking and the pair

`(term(l), type_prefix(l))`

at the moment of passing to `More`, gets value of

`conres(term, typeprefix, CS)`

else backtracking will occur before `p` reaches `More`.

Proof. The proof follows by induction on the size of the involved terms. Observe that similar to the Putting Lemma no real variable creation occurs: When an auxiliary variable X_k (generated by nf_a) is created on the heap via `unify_variable` in `Write` mode, its `<VAR,TOP>` initialization will be overwritten by a subsequent `get_structure` instruction corresponding to the subterm represented by X_k. Note also that if `CS` is solvable, then `conres(term, typeprefix, CS)` $\neq$ `nil` because `CS` $\cup$ `typeprefix` is also solvable since the intersection between `typeprefix` and any `type_prefix(`y_n`)` is already contained in `CS`. $\square$

In order to uphold the

HEAP VARIABLES CONSTRAINT: No heap variable points outside the heap, i.e. for any `l` $\in$ **HEAP** with `boh` $\leq$ `l` $<$ `h` and `tag(l)` = `REF`, we have `boh` $\leq$ `ref(l)` $<$ `h`.

the instruction `unify_local_value` in `Write` mode creates a new heap variable for a so-called local variable (cf. B.5):

`local(l)` $\equiv$ `unbound(l) & l` $\in$ **HEAP** `& NOT(boh` $\leq$ `l` $<$ `h)`

For a discussion of local variables see [AK91] or [BR92b]. In the PROTOS-L case the type restriction of the local variable must be taken into account which

is done by the binding update in our `mk_heap_variable` abbreviation. Thus, the HEAP VARIABLES CONSTRAINT as well as the

HEAP VARIABLES LEMMA: If the `put_code` and `get_code` functions generate `unify_local_value` instead of `unify_value` for all occurrences of local variables, then the execution of `put_seq` and `get_seq` preserve the HEAP VARIABLES CONSTRAINT [BR92b].

carries over to the PROTOS-L case, provided we ensure

BINDING CONDITION 2: The `bind` update preserves the HEAP VARIABLES CONSTRAINT.

3.5. Putting of Constraints

In this section we will still keep the type constraint representation abstract, while specifying the conditions about the constraint handling code (for realization of `add_constraint` of Section 2) in order to prove a theorem corresponding to the Pure Prolog Theorem of [BR92b] (see 4).

The compile function will be refined using

$$\texttt{put\_constraint\_seq}(\{\texttt{Y}_1:\texttt{tt}_1,\ldots,\texttt{Y}_r:\texttt{tt}_r\}) = [\texttt{put\_constraint}(\texttt{y}_1,\texttt{tt}_1),\ \ldots,\ \texttt{put\_constraint}(\texttt{y}_r,\texttt{tt}_r)]$$

for which we use the new instruction `put_constraint(`$\texttt{y}_n$`,tt)` (where `tt` $\in$ **TYPETERM**) and the following rule:

Put-Constraint

```
if   RUN
  & code(p) = put_constraint(l,tt)
then
  insert_type(l,tt)
  succeed
```

The update for inserting a type restriction has still the straightforward definition given in 3.1 (i.e. `ref(l) := tt`), but will be refined later when we introduce a representation of type terms. In any case it must satisfy the following

TYPE INSERTING CONDITION: For any $\texttt{l}_1$, `l` $\in$ **DATAARRA**, with `term` resp. `term'` values of `term(l)` and with `prefix` resp. `prefix'` values of `type_prefix(l)` before resp. after execution of `insert_type(`$\texttt{l}_1$`,tt)` we have if `unbound(`$\texttt{l}_1$`)` holds:

`(term', prefix') = conres(term,prefix\{mk_var(`$\texttt{l}_1$`)},{mk_var(`$\texttt{l}_1$`):tt})`

For the definition given above the type inserting condition is obviously satisfied.

4. PAM Algebras

4.1. Environment and Choicepoint Representation

The stack of states and environments of PROTOS-L algebras with compiled AND/ OR structure of Section 2 are now represented by a subalgebra of **DATAAREA**

(**STACK**; bos; +, -; val)

with bos $\in$ **STACK** representing the bottom element corresponding to nil in Section 2. The concrete memory layout can be done as in the WAM [BR92b] (see Appendix B.6) since the only type-related action is in the allocation of n free variable cells in the rule for Allocate: This situation is covered by our modified mk_unbound abbreviation that assigns the trivial TOP type restriction to each such initialized variable:

allocate

```
if   OK
  & code(p) = allocate(n)
then
  e := tos(b,e)
  val(ce(tos(b,e))) := e
  val(cp(tos(b,e))) := cp
  FORALL i = 1,...,n DO
     mk_unbound(y_i(tos(b,e)))
  ENDFORALL
  succeed
```

deallocate

```
if   OK
  & code(p) = deallocate
then
  e := val(ce(e))
  cp := val(cp(e))
  succeed
```

4.2. Trailing

As is standard practice in the WAM, we assume that **HEAP** $<$ **STACK** $<$ **AREGS** and the WAM binding discipline:

BINDING CONDITION 3: If unbound(l_1) and unbound(l_2) and bind(l_1,l_2) does not initiate backtracking, then after executing bind(l_1,l_2) the higher location will be bound to the lower one.

Together, these conditions imply BINDING CONDITION 2 and also the

STACK VARIABLES PROPERTY: Every stack variable l points either to the heap or to a lower location of the stack, i.e. ref(l) $\in$ **HEAP** with boh $\leq$ l $<$ h, or ref(l) $\in$ **STACK** with bos $\leq$ l $\leq$ tos(b,e).

Whereas BINDING CONDITION 3 and the STACK VARIABLES PROPERTY are exactly as in the WAM case [BR92b], for trailing variable bindings also the type restrictions must be taken into account in the PAM. Since variables in the PAM carry a type restriction represented in the ref value of a location - which is updated when binding the variable -, the type restriction must be saved upon binding and recovered upon backtracking. Strictly speaking, it would be sufficient to save only the ref value of a location; however, for use in a later refinement -when we will introduce different tags for free variables - we also trail the tag of a location. Therefore, in the **DATAAREA** subalgebra

(**TRAIL**, tr, botr; +, -; ref¨)

with tr, botr $\in$ **TRAIL** being the top and bottom elements, the codomain of the function

ref¨: **TRAIL** $\rightarrow$ **DATAAREA** $\times$ **PO**

records also the complete val decoration. The trail update, to be executed when changing the value of a location l during binding is then:

```
trail(l) ≡ ref¨(tr) := (l, val(l))
           tr := tr+
```

Note that this is a non-optimized version of the trailing operation; we could have also used a conditional trailing governed by the condition `boh ≤ l < h & l < hb OR bos ≤ l ≤ tos(b,e) & l < b`.

For `t ∈` **TRAIL** with `ref¨(t) = (l, v)` we use the following abbreviation for the two obvious projections on `ref¨(t)`:

```
location(t) ≡ l          value(t) ≡ v
```

Upon backtracking we must now unwind the trail

```
backtrack    ≡ p := val(p(b))
               unwind_trail
unwind_trail ≡ FORALL t = tr-,...,tr(b) DO
                  location(t) ← value(t)
               ENDFORALL
```

where `value(t)` retrieves the previous tag and type restriction of `location(t)`. We still leave the binding update abstract, but pose the following

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

Note that due to the update on the type restrictions of a variable the trailing of *both* locations l_1 and l_2 may be triggered by `bind(`l_1`,`l_2`)`; moreover, if e.g. l_2 denotes a polymorphic term containing variables these variables also have to be trailed if they get another type restriction in the binding process (see [BB96]).

4.3. Pure PROTOS-L theorem

In order to establish a correctness proof of compilation to PAM algebras developed so far from PROTOS-L algebras with compiled AND/OR structure of Section 2, we can generalize the "Pure Prolog Theorem" of [BR92b] to our case. We will thus construct a function $\mathcal{F}$ (c.f. Section 1) from the PROTOS-L algebras to the PAM algebras. We will also first ignore cutpoints (`ct, ct'`) which are not needed for pure PROTOS-L, as well as variable renaming indices (`vi, vi'`) since as in the WAM case the renaming is ensured by the offsets in the stack and the heap. Further, all names of universes and functions on Section 2 will get an index 1. For the function `compile` dealing with the term representing algebras we have

```
compile({P} H <-- G1 & ... & Gn) =
            flatten([allocate(r), put_constraint_seq(P),
get_seq(H), call_seq(G1), ..., call_seq(G1),
                      deallocate, proceed])
```

The abstraction function $\mathcal{F}$ maps PAM rules to PROTOS-L rules in the obvious way. It is defined via a mapping between instruction sequences (which directly correspond to rule sequences). For instance, with respect to unification and type constraint solving we have

```
call_seq(G)           ↦ call(G)
get_seq(H)            ↦ unify(H)
put_constraint_seq(P) ↦ add_constraint(P)
```

This correspondence also defines a (partial) function

$$\texttt{codepointer}: \mathbf{CODEAREA} \times \mathbf{CODEAREA}_1$$

by mapping e.g. the beginning of `get_seq(H)` to the location labelled with `unify(H)`. Furthermore, we establish the functions

$$\begin{array}{ll}
\texttt{css}: & \mathbf{TRAIL} \to \mathbf{CSS}_1 \\
\texttt{subst}: & \mathbf{TRAIL} \to \mathbf{SUBST}_1 \\
\texttt{choicepoint}: & \mathbf{STACK} \to \mathbf{STATE}_1 \\
\texttt{env}: & \mathbf{STACK} \to \mathbf{ENV}_1 \\
\texttt{term}: & \mathbf{DATAAREA} \times \mathbf{TRAIL} \to \mathbf{TERM}_1 \\
\texttt{typeprefix}: & \mathbf{DATAAREA} \times \mathbf{TRAIL} \to \mathbf{TYPEPREFIX}_1
\end{array}$$

where we have added - w.r.t. the WAM case in [BR92b] - the functions `css` and `typeprefix` in order to construct the correspondence between the constraint representations. Viewing an element of $\mathbf{STATE}_1$ (resp. $\mathbf{ENV}_1$) as a tuple of its `cs, p, cp, e, b` (resp. cp', ce) values, these functions are defined by:

$\texttt{term(l,}l_t\texttt{)}$ yields the value `term(l)` would take after having unwound the trail down to l_t

$\texttt{typeprefix(l,}l_t\texttt{)}$ yields the value `type_prefix(l)` would take after having unwound the trail down to l_t

$$\texttt{css}(l_t) = \bigcup_{botr \le l < tr} \{\texttt{mk\_var(location(l))} \doteq \texttt{term(location(l),}l_t\texttt{)}\} \cup \texttt{typeprefix(location(l),}l_t\texttt{)}$$

$$\texttt{subst}(l_t) = \texttt{subst\_part}(\texttt{css}(l_t))$$

$$\begin{array}{l}
\texttt{choicepoint(lb)} = \langle\ \texttt{css(val(tr(lb))),} \\
\qquad\qquad\qquad\quad \texttt{codepointer(val(p(lb))),} \\
\qquad\qquad\qquad\quad \texttt{codepointer(val(cp(lb))),} \\
\qquad\qquad\qquad\quad \texttt{env(val(e(lb))),} \\
\qquad\qquad\qquad\quad \texttt{choicepoint(val(b(lb)))}\ \rangle
\end{array}$$

$$\begin{array}{l}
\texttt{env(le)} = \langle\ \texttt{codepointer(val(cp(le))),} \\
\qquad\qquad\quad \texttt{env(val(ce(le)))}\ \rangle
\end{array}$$

The 0-ary functions are defined by

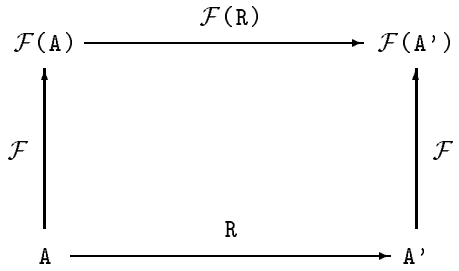
$$\begin{array}{ll}
\texttt{cs}_1 & = \texttt{css(tr)} \\
\texttt{p}_1 & = \texttt{codepointer(p)} \\
\texttt{cp}_1 & = \texttt{codepointer(cp)} \\
\texttt{e}_1 & = \texttt{env(e)} \\
\texttt{b}_1 & = \texttt{choicepoint(b)}
\end{array}$$

Furthermore, for the current activator $\texttt{act}_1$ of 2.4 we have for `code(cp-)` = `call(g,m,r)` the correspondence

$$\texttt{act}_1 = \texttt{g(term(x}_1\texttt{),}\ldots\texttt{,term(x}_m\texttt{))}$$

Correctness Theorem 1 (PURE PROTOS-L THEOREM): Compilation from the PROTOS-L algebras with compiled AND/OR structure (of Section 2) to the PAM algebras developed so far (and thus satisfying all the conditions explicitely stated above) is correct.

Proof. For the proof it suffices to show that for any PAM algebra `A` and any transition rule sequence `R` such that $\mathcal{F}$`(A)` and $\mathcal{F}$`(R)` is defined, the diagram



commutes. This follows by case analysis, relying on the conditions and lemmas established so far. In particular, w.r.t. type constraints we observe the fact that `allocate` allocates a new variable location (with `TOP` restriction) for every variable occurring in the clause. These locations are used by the `put_constraint` instructions, so that the preconditions for the TYPE INSERTING CONDITION hold. □

5. Additional WAM optimizations in the PAM

5.1. Environment Trimming and Last Call Optimization

Environment trimming and last call optimization (LCO) are among the most prominent optimizations in the WAM; for a discussion we refer to [AK91] and [BR92b]. The necessary ARGUMENT REGISTERS PROPERTY as formulated in [BR92b] can be ensured by the compiler by generating a `put_unsafe_value(`$\mathtt{y}_n$`,`$\mathtt{x}_j$`)` instruction instead of `put_value(`$\mathtt{y}_n$`,`$\mathtt{x}_j$`)` for each unsafe occurrence of $\mathtt{Y}_n$. This instruction is executed by the rule:

Put-Unsafe-Value

```
if   RUN
  & code(p) = put_unsafe_value(y_n,x_j)
  & deref(y_n) ≤ e  | deref(y_n) > e
then
  x_j ← deref(y_n)   | mk_heap_var(deref(y_n))
                     | x_j ← <REF,h>
  succeed
```

Note that the condition `deref(`$\mathtt{y}_n$`) > e` implies `unbound(deref(`$\mathtt{y}_n$`))`. Thus, in case of $\mathtt{y}_n$ being unsafe, a new variable is created on the heap, referenced by both $\mathtt{y}_n$ and $\mathtt{x}_j$. Unlike in Prolog, in PROTOS-L the type restriction of $\mathtt{y}_i$ must be copied to the new heap variable - this is already taken into account by the bind update in our `mk_heap_var` abbreviation introduced in Section 3.4. Therefore, following the argumentation in [BR92b], we can savely assume that the compiler enforces environment trimming and also last call optimization (LCO). Thus, every `call` instruction gets an additional parameter `n` where `n` is the number of variables that are still needed in the environment. LCO then means that the instruction sequence

```
Call(g,a,0), Deallocate, Proceed
```

is replaced by

```
Deallocate, Execute(g,a)
```

which disregards the current environment *before* calling the last subgoal of a clause.

5.2. Initializing Temporary and Permanent Variables

Up to now, when allocating an environment, we have allocated `r` value cells in that environment, where `r` is the number of variables occurring in the clause. A sequence of `r` corresponding `put_constraint(`y_j`,`tt_j`)` instructions initialized the type restriction on the variables y_j to tt_j found in the clause's type prefix.

However, as explained in [BMS91], the *first* occurrence of a variable in a PROTOS-L clause is sufficient to consider the statically available type restriction. (The specialized instructions of [BMS91, BM94] for variables with monomorphic, polymorphic, or with no type restriction are introduced as an optimization in [BB96].) Both temporary and permanent variables can be initialized "on the fly"; for a discussion of the classification of variables into temporary and permanent ones which was introduced by [War83] we refer to [AK91] and [BR92b]. Thus, we modify our compile function such that for a temporary variable, Y_n, y_n is replaced by fresh X_i, x_i, and such that

```
get_variable        put_variable        unify_variable
```

instructions are generated for the *first* occurrence of a variable, replacing the so-far used `get_value`, `put_value` (resp. `put_unsafe_value`), and `unify_value` instructions, respectively.

Put-1 (X variable)

if RUN
& code(p) = put_variable(x_i,x_j,tt)
then
mk_unbound(h,tt)
x_i ← <REF,h>
x_j ← <REF,h>
succeed

Put-2 (Y variable)

if RUN
& code(p) = put_variable(y_n,x_j,tt)
then
mk_unbound(y_n,tt)
x_j ← <REF,y_n>
succeed

When initializing a temporary variable with `put_variable`, a new heap cell must be allocated, which is not the case when initializing a permanent variable, provided that `put_unsafe_variable` and `unify_local_value` instructions are used properly. This, however, has already been verified (see Section 5.1). In both cases, the `mk_unbound(l,tt)` update corresponds to the `mk_unbound(l)` update for that variable carried out previously during allocation, and the `insert_type(l,tt)` update carried out by the `put_constraint` instruction immediately after allocation (c.f. 3.5). Therefore, since the `put_variable` instruction corresponds to the *first* occurrence of the variable X_i resp. Y_n, we can savely drop its initialization during allocation and its complete `put_constraint` instruction.

get_variable

```
if   RUN
  & code(p) = get_variable(l,xj,tt)
then
  mk_unbound(l,tt)
  bind(l,xj)
  succeed
```

Whereas in the WAM case the `get_variable` instruction always succeeds, in the PROTOS-L case we have to check that the clause's type restriction `tt` for x_j is satisfied. This is achieved by setting `l` to an unbound variable, inserting the type term `tt` as its type restriction, and binding `l` and x_j. The latter is sufficient as the binding update will do the binding only if the type restrictions are satisfied; otherwise it will fail and initiate backtracking (c.f. the BINDING CONDITION of Section 3.2).

unify_variable

```
if   RUN
  & code(p) = unify_variable(l,tt)
  & mode = Read       | mode = Write
then
  mk_unbound(l,tt)    | mk_unbound(h,tt)
  bind(l,nextarg)     | l ← <REF,h>
  nextarg := nextarg+ | h := h+
  succeed
```

The instruction `unify_variable` in `Read` mode has to make sure that the incoming argument satisfies the type restriction, which - as in `get_variable` - is achieved by a `bind` update. In `Write` mode, the type restriction has just to be inserted into a new heap cell.

As argued above for `put_variable`, the initialization of a free value cell during allocation as well as the `put_constraint` instruction can also be dropped for all variables initialized by `get_variable` or `unify_variable`, which leads us to the

INITIALIZATION LEMMA: Given `l` > `e`, the instruction `put_variable(l,`x_j`,tt)` (`get_variable(l,`x_j`,tt)`, `unify_variable(l,tt)`, resp.) is equivalent to initializing `l` to unbound with `mk_unbound(l)`, executing `put_constraint(l,tt)`, and then executing `put_unsafe_value(l,`x_j`)` (`get_value(l,`x_j`)`, `unify_local_value(l)`, resp.). For a permanent variable Y_n, the instruction `put_variable(`y_n`,`x_j`,tt)` is equivalent to initializing y_n to unbound with `mk_unbound(`y_n`)`, executing `put_constraint(`y_n`,tt)`, and then executing `put_value(`y_n`,`x_j`)`.

Thus, the rule for `allocate` looses its initialization update, and the compile function is modified such that no `put_constraint` instruction is generated any more. Moreover, the argumentation of Section3.2 and 3.2 of [BR92b] can be applied to our modified setting, implying also the correctness of special compilation of facts and chain rules where no environment needs to be allocated at all.

5.3. Switching instructions and the Cut

The PAM contains all switching instructions known from the WAM, and since no type specific considerations have to be taken into account, their treatment

in the evolving algebra approach in [BR92b] carries over to the PAM as well. Thus, compared to the compiled AND/OR structure (Sect. 2 and Appendix A) the indexing and choicepoint handling rules now also get the predicate arity `n` as an additional parameter, and the choicepoint information is not attached to a newly created stack element, but by reusing and "overwriting" elements on the stack (see B.7). However, in PROTOS-L additionally a switch on the *type restriction* of a variable is possible (see [BB96]).

For the establishment of the Pure PROTOS-L Theorem we had deliberately left out the built-in predicate `cut`. Since there is no interdependence between `cut` and the type constraints of PROTOS-L, the cut treatment of Prolog carries over to our case as well [BR92a]: We could either extend every environment by a cutpointer, to be set and restored just as in Section 2, or we could allocate an extra (permanent) variable in those environments containig a so-called *deep cut*. This extra variable would then be set immediately after allocation, and its value would be assigned to the backtracking pointer `b` when a `cut` is encountered (see also [AK91]).

5.4. Main Theorem

Putting everything together developed so far, we obtain

Correctness Theorem 2 (Main Theorem): Compilation from PROTOS-L algebras to the PAM algebras developed so far is correct. Thus, since we kept the notion of types abstract, for every such type-constraint logic programming system L and for every compiler satisfying the specified conditions, compilation to the WAM extension with this abstract notion of types is correct.

Thus, any type system satisfying the minimal preconditions on the `solution` function stated in Section 2.1 is covered by the development above.

References

[AK91] H. Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction.* MIT Press, Cambridge, MA, 1991.

[BB91] C. Beierle and E. Börger. A WAM extension for type-constraint logic programming: Specification and correctness proof. IWBS Report 200, IBM Germany, Scientific Center, Inst. for Knowledge Based Systems, Stuttgart, 1991.

[BB92] 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.* LNCS 626. Springer-Verlag, Berlin, 1992.

[BB96] C. Beierle and E. Börger. Refinement of a typed WAM extension by polymorphic order-sorted types. *Formal Aspects of Computing*, 8(5), 1996 (to appear).

[BBM91] C. Beierle, S. Böttcher, and G. Meyer. Draft report of the logic programming language PROTOS-L. IWBS Report 175, IBM Germany, Stuttgart, 1991. Revised version: Working Paper 4, IBM Germany, Scientific Center, Institute for Logics and Linguistics, Heidelberg, July 1994.

[Bei92] C. Beierle. Logic programming with typed unification and its realization on an abstract machine. *IBM Journal of Research and Development*, 36(3):375–390, May 1992.

[Bei94] C. Beierle. Formal design of an abstract machine for constraint logic programming. In B. Pehrson and I. Simon, editors, *Technology and Foundations - Proceedings of the IFIP Congress 94*, volume 1, pages 377–382. Elsevier / North Holland, Amsterdam, 1994.

[Bei95] C. Beierle. Concepts, implementation, and applications of a typed logic programming language. In C. Beierle and L. Plümer, editors, *Logic Programming: Formal Methods and Practical Applications*, Studies in Computer Science and Artificial Intelligence, chapter 5, pages 139–167. Elsevier Science B.V./North-Holland, Amsterdam, 1995.

[BM94] C. Beierle and G. Meyer. Run-time type computations in the Warren Abstract Machine. *Journal of Logic Programming*, 18(2):123–148, February 1994.

[BMS91] C. Beierle, G. Meyer, and H. Semle. Extending the Warren Abstract Machine to polymorphic order-sorted resolution. In V. Saraswat and K. Ueda, editors, *Logic Programming: Proceedings of the 1991 International Symposium*, pages 272–286, Cambridge, MA, 1991. MIT Press.

[Bör90a] 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*. LNCS 440, pages 36–64. Springer-Verlag, Berlin, 1990.

[Bör90b] E. Börger. A logical operational semantics of full Prolog. Part II. Built-in predicates for database manipulations. In B. Rovan, editor, *MFCS'90 - Mathematical Foundations of Computer Science*. LNCS 452, pages 1–14, Berlin, 1990. Springer-Verlag.

[BR91] E. Börger and D. Rosenzweig. From Prolog algebras towards WAM - a mathematical study of implementation. In E. Börger, H. Kleine Büning, M. M. Richter, and W. Schönfeld, editors, *Computer Science Logic*. LNCS 533, pages 31–66. Springer-Verlag, Berlin, 1991.

[BR92a] E. Börger and D. Rosenzweig. The WAM - definition and compiler correctness. TR-14/92, Dipartimento di Informatica, Università di Pisa, 1992. (Revised version in: C. Beierle, L. Plümer (Eds.), *Logic Programming: Formal Methods and Practical Applications*, North-Holland, 1995).

[BR92b] E. Börger and D. Rosenzweig. WAM algebras - a mathematical study of implementation, Part II. In A. Voronkov, editor, *Logic Programming*. LNAI 592, pages 35–54, Berlin, 1992. Springer-Verlag.

[BS91] E. Börger and P. H. Schmitt. A formal operational semantics for languages of type Prolog III. In E. Börger, H. Kleine Büning, M. M. Richter, and W. Schönfeld, editors, *Computer Science Logic*. LNCS 533, pages 67–79. Springer-Verlag, Berlin, 1991.

[BS95] E. Börger and R. Salamone. CLAM specification for provably correct compilation of CLP(R) programs. In E. Börger, editor, *Specification and Validation Methods*, pages 97–130. Oxford University Press, 1995.

[Col90] A. Colmerauer. An introduction to Prolog-III. *Communications of the ACM*, 33(7):69–906, July 1990.

[EM89] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 2*. EATCS Monographs on Theoretical Computer Science. Springer-Verlag, Berlin, 1989.

[GM86] J. A. Goguen and J. Meseguer. Eqlog: Equality, types, and generic modules for logic programming. In D. DeGroot and G. Lindstrom, editors, *Logic Programming, Functions, Relations, and Equations*, pages 295–363. Prentice Hall, 1986.

[Gur88] Y. Gurevich. Logic and the challenge of computer science. In E. Börger, editor, *Trends in Theoretical Computer Science*, pages 1–57. Computer Science Press, 1988.

[Gur91] Y. Gurevich. Evolving algebras. A tutorial introduction. *EATCS Bulletin*, 43, February 1991.

[Han88] M. Hanus. *Horn Clause Specifications with Polymorphic Types*. PhD thesis, FB Informatik, Universität Dortmund, 1988.

[Han91] M. Hanus. Horn clause programs with polymorphic types: Semantics and resolution. *Theoretical Computer Science*, 89:63–106, 1991.

[JL87] 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.

[JMSY90] J. Jaffar, S. Michaylov, P. Stuckey, and R. Yap. The CLP($\mathcal{R}$) language and system. Technical Report RC 16292, IBM Research Devision, 1990.

[MO84] A. Mycroft and R. A. O'Keefe. A polymorphic type system for Prolog. *Artificial Intelligence*, 23:295–307, 1984.

[NM88] G. Nadathur and D. Miller. An overview of λProlog. In K. Bowen and R. Kowalski, editors, *Proceedings of the Fifth International Conference and Symposium on*

Logic Programming, pages 810–827, Cambridge, MA, 1988. MIT Press.

[Rus92] D. M. Russinoff. A verified Prolog compiler for the Warren Abstract Machine. *Journal of Logic Programming*, 13:367–412, 1992.

[Smo88] G. Smolka. TEL (Version 0.9), Report and User Manual. SEKI-Report SR 87-17, FB Informatik, Universität Kaiserslautern, 1988.

[Smo89] G. Smolka. *Logic Programming over Polymorphically Order-Sorted Types*. PhD thesis, FB Informatik, Univ. Kaiserslautern, 1989.

[War83] D. H. D. Warren. An Abstract PROLOG Instruction Set. Technical Report 309, SRI, 1983.

A. Transition rules for compiled And/Or structure

allocate

```
if   OK
  & code(p) = allocate
then
  PUSH_ENV temp IN
     cp'(temp) := cp
     vi'(temp) := vi
     ct'(temp) := ct
  ENDPUSH
  succeed
```

deallocate

```
if   OK
  & code(p) = deallocate
then
  POP_ENV
  cp := cp'(e)
  succeed
```

call

```
if   OK
  & code(p) = call(G)
  & is_user_defined(G)
then
  let p1 = procdef(act,cs,prog)
  if code(p1) = fail
  then backtrack
  else  p := p1
        ct := b
        cp := p+
```

unify

```
if   OK
  & code(p) = unify(H)
then
  let cs1 = {act ≐ rename(H,vi)}
  if solvable(cs ∪ cs1)
  then cs := cs ∪ cs1
       vi := vi + 1
       succeed
  else  backtrack
```

true/fail/cut

```
if  OK
  & code(p) = call(BIP)
  & BIP =
    true |fail      |cut
then
  succeed|backtrack|b := ct'(e)
         |          |succeed
```

add_constraint

```
if   OK
  & code(p) = add_constraint(P)
then
  let cs1 = rename(P,vi)
  if solvable(cs ∪ cs1)
  then cs := cs ∪ cs1
       succeed
  else  backtrack
```

try_me_else/try

```
if   OK
  & code(p) =
    try_me_else(N)|try(L)
then
  PUSH_STATE temp IN
    store_state_in(temp)
    p(temp) := N  |p(temp) := p+
    p:= p+        |p := L
  ENDPUSH
```

trust_me/trust

```
if   OK
  & code(p) =
     trust_me | trust(L)
then
  fetch_state_from(b)
  POP_STATE
  p:= p+       | p := L
```

retry_me_else/retry

```
if   OK
  & code(p) =
    retry_me_else(N)|retry(L)
then
  fetch_state_from(b)
  p(b) := N          |p(b) := p+
  p:= p+             |p := L
```

proceed

```
if   OK
  & code(p) = proceed
  & code(cp)
     = proceed | ≠ proceed
then
  stop := 1     | p := cp
```

switch_on_structure

```
if  OK
  & code(p) =
     switch_on_structure(i,T)
then
  let x_i = arg(act,i)
  p := select(T,func(x_i),
              arity(x_i))
```

switch_on_term

```
if  OK
  & code(p) =
     switch_on_term(i,Lv,Ls)
  & let x_i = arg(act,i)
    is_var(x_i) | is_struct(x_i)
then
    p := Lv    | p := Ls
```

Abbreviations:

```
succeed ≡ p := p + 1

OK      ≡ stop = 0

backtrack ≡ if b = nil
            then stop := -1
            else p := p(b)

PUSH_STATE temp IN
        updates ENDPUSH ≡
  EXTEND STATE BY temp WITH
     b := temp
     b(temp) := b
     temp- := tos(b,e)
     updates
  ENDEXTEND

PUSH_ENV temp IN
        updates ENDPUSH ≡
  EXTEND ENV BY temp WITH
     e := temp
     ce(temp) := e
     temp- := tos(b,e)
     updates
  ENDEXTEND

POP_STATE ≡ b := b(b)

POP_ENV ≡ e := ce(e)

fetch_state_from(b) ≡
   cs := cs(b)
   cp := cp(b)
   e  := e(b)

store_state_in(temp) ≡
   cs(temp) := cs
   cp(temp) := cp
   e(temp)  := e
```

B. Transition rules for the PAM with abstract type terms

B.1. Low level unification

Unify-1 (success)

```
if   OK & what_to_do = Unify
  & pdl = nil
then
  what_to_do := Run
```

Unify-2 (Unify-Var-Any)

```
if   UNIF
  & unbound(dl) | NOT(unbound(dl))
                | & unbound(dr))
then
  bind(dl,dr) | bind(dr,dl)
  pdl := pdl--
```

Unify-3 (Unify-Struc-Struc)

```
if   UNIF
  & NOT( unbound(dl) or unbound(dr) )
  & val(ref(dl)) = val(ref(dr))
then
  FORALL i = 1,...,arity(val(ref(dl))) DO
   ref'(pdl+2*arity(val(ref(dl)))-2*i) := ref(dl)+i
   ref'(pdl+2*arity(val(ref(dl)))-2*i-1) := ref(dr)+i
  ENDFORALL
  pdl := pdl+2*arity(val(ref(dl)))-2
```

Unify-4 (Unify-Struc-Struc)

```
if   UNIF
  & NOT( unbound(dl) or unbound(dr) )
  & NOT( val(ref(dl)) = val(ref(dr)) )
then
  backtrack
  what_to_do := Run
```

Abbreviations:

```
dr   ≡ deref(right)
dl   ≡ deref(left)
UNIF ≡ OK & what_to_do = Unify
RUN  ≡ OK & what_to_do = Run
```

B.2. Putting and Getting Code

The code for putting (resp. getting) instructions corresponding to a body goal (resp. the clause head) is defined using the *term normal form* of first order logic. Its two froms nf_s (resp. nf_a) correspond to the synthesis (resp. analysis) of terms:

$$
\begin{array}{rcl}
nf(\mathtt{X}_i{=}\mathtt{Y}_n) & = & [\mathtt{X}_i{=}\mathtt{Y}_n] \\
nf(\mathtt{Y}_i{=}\mathtt{Y}_n) & = & [\ ] \\
nf_s(\mathtt{X}_i{=}\mathtt{f}(\mathtt{s}_1,\ldots,\mathtt{s}_m)) & = & \mathtt{flatten}([nf_s(\mathtt{Z}_1{=}\mathtt{s}_1),\ldots,nf_s(\mathtt{Z}_m{=}\mathtt{s}_m), \\
 & & \qquad \mathtt{X}_i{=}\mathtt{f}(\mathtt{Z}_1,\ldots,\mathtt{Z}_m)]) \\
nf_a(\mathtt{X}_i{=}\mathtt{f}(\mathtt{s}_1,\ldots,\mathtt{s}_m)) & = & \mathtt{flatten}([\mathtt{X}_i{=}\mathtt{f}(\mathtt{Z}_1,\ldots,\mathtt{Z}_m), \\
 & & \qquad nf_a(\mathtt{Z}_1{=}\mathtt{s}_1),\ldots,nf_a(\mathtt{Z}_m{=}\mathtt{s}_m)])
\end{array}
$$

The function `put_instr` (resp. `get_instr`) of a normalized equation is defined by the following table, where `j` stands for an arbitrary 'top level' index (corresponding to the input $\mathtt{X}_i{=}\mathtt{t}$ for term normalization) and `k` for a 'non top level' index (corresponding to an auxiliary variable introduced by normalization itself):

$$
\begin{array}{rcl}
\mathtt{X}_j{=}\mathtt{Y}_n & \rightarrow & [xxx\mathtt{\_value}(\mathtt{y}_n,\mathtt{x}_j] \\
\mathtt{X}_k{=}\mathtt{Y}_n & \rightarrow & [\mathtt{unify\_value}(\mathtt{y}_n)] \\
\mathtt{X}_i{=}\mathtt{f}(\mathtt{Z}_1,\ldots,\mathtt{Z}_a) & \rightarrow & [xxx\mathtt{\_structure}(\mathtt{entry}(\mathtt{f},\mathtt{a}),\mathtt{x}_i), \\
 & & \quad \mathtt{unify}_{xxx}(\mathtt{z}_1),\ldots,\mathtt{unify}_{xxx}(\mathtt{z}_a)]
\end{array}
$$

where *xxx* stands for `put` (resp. `get`), $\mathtt{y}_i \in$ **DATAAREA**, $\mathtt{x}_i \in$ **AREGS**, and with

$$
\mathtt{unify}_{xxx}(\mathtt{z}_i) \quad = \quad \begin{cases}
\mathtt{unify\_value}(\mathtt{Y}_n) & \text{if } \mathtt{Z}_i = \mathtt{Y}_n \text{ and } xxx = \mathtt{put} \\
\mathtt{unify\_value}(\mathtt{X}_k) & \text{if } \mathtt{Z}_i = \mathtt{X}_k \text{ and } xxx = \mathtt{put} \\
\mathtt{unify\_value}(\mathtt{y}_n) & \text{if } \mathtt{Z}_i = \mathtt{Y}_n \text{ and } xxx = \mathtt{get} \\
\mathtt{unify\_variable}(\mathtt{X}_k) & \text{if } \mathtt{Z}_i = \mathtt{X}_k \text{ and } xxx = \mathtt{get}
\end{cases}
$$

The function `put_code` (resp. `get_code`) is defined by flattening the result of mapping `put_instr` (resp. `get_instr`) along $nf_a(\mathtt{X}_i{=}\mathtt{t})$ (resp. $nf_s(\mathtt{X}_i{=}\mathtt{t})$). The function `put_seq` (resp. `get_seq`) specifies how a body goal (resp. clause head) of the form $\mathtt{g}(\mathtt{s}_1,\ldots,\mathtt{s}_m)$ is compiled:

$$
xxx\mathtt{\_seq}(\mathtt{g}(\mathtt{s}_1,\ldots,\mathtt{s}_m)) \quad = \quad \mathtt{flatten}([xxx\mathtt{\_code}(\mathtt{X}_1{=}\mathtt{s}_1),\ldots, \\
xxx\mathtt{\_code}(\mathtt{X}_m{=}\mathtt{s}_m)])
$$

with 'top level' $\mathtt{j} = 1,\ldots,\mathtt{m}$.

Additionally, for the HEAP VARIABLES LEMMA and the proof of the "Pure PROTOS-L theorem" in 4 we assume that the `put_code` and `get_code` functions generate `unify_local_value` instead of `unify_value` for all occurrences of *local* variables, and that

$$
\mathtt{call\_seq}(\mathtt{g}(\mathtt{s}_1,\ldots,\mathtt{s}_k)) \quad = \quad \mathtt{flatten}([\mathtt{put\_seq}(\mathtt{g}(\mathtt{s}_1,\ldots,\mathtt{s}_k)), \\
\mathtt{call}(\mathtt{g},\mathtt{k},\mathtt{r})])
$$

with $\{\mathtt{Y}_1,\ldots,\mathtt{Y}_r\}$ being all variables occurring in the clause.

Additional compiler assumptions are given in Section 5 for the optimizations introduced there (environment trimming, LCO, variable initialization "on the fly", etc.).

B.3. Putting of terms

put_value

if `RUN`
`&` $\mathtt{code(p)} = \mathtt{put\_value}(\mathtt{l},\mathtt{x}_j)$
then
$\mathtt{x}_j \leftarrow \mathtt{l}$
`succeed`

Put-Unsafe-Value

```
if   RUN
  & code(p) = put_unsafe_value(y_n,x_j)
  & deref(y_n) ≤ e  | deref(y_n) > e
then
  x_j ← deref(y_n)   | mk_heap_var(deref(y_n))
                     | x_j ← <REF,h>
  succeed
```

put_structure

```
if   RUN
  & code(p) = put_structure(f,x_i)
then
  h ← <STRUC,h+>
  x_i ← <STRUC,h+>
  val(h+) := f
  h := h++
  mode := write
  succeed
```

"On the fly" initialization (Sec. 5.2):

Put-1 (X variable)

```
if   RUN
  & code(p) =
       put_variable(x_i,x_j,tt)
then
  mk_unbound(h,tt)
  x_i ← <REF,h>
  x_j ← <REF,h>
  succeed
```

Put-2 (Y variable)

```
if   RUN
  & code(p) =
       put_variable(y_n,x_j,tt)
then
  mk_unbound(y_n,tt)
  x_j ← <REF,y_n>
  succeed
```

B.4. Getting of terms

get_value

```
if   RUN
  & code(p) = get_value(l,x_j)
then
  unify(l,x_j)
  succeed
```

Get-Structure-1

```
if   RUN
  & code(p) = get_structure(f,x_i)
  & tag(deref(x_i)) = STRUC
  & val(ref(deref(x_i))) = f   | val(ref(deref(x_i))) ≠ f
then
  nextarg := ref(deref(x_i))+  | backtrack
  mode := Read                 |
  succeed                      |
```

Get-Structure-2

```
if   RUN
  & code(p) = get_structure(f,x_i)
  & unbound(deref(x_i))
```

```
  & can_propagate(f,ref(deref(x_i)))
              = true                                  | = false
  & trivially_propagates(f,ref(deref(x_i)))           |
      = true    | = false                             |
then
  h ← <STRUC,h+>                                      | backtrack
  bind(deref(x_i),h)                                  |
  val(h+) := f                                        |
  h := h++                                            |
  mode := Write | nextarg := h++                       |
                | mk_unbounds(h+,propagate_list(f,ref(deref(x_i))) |
                | mode := Read                        |
  succeed                                             |
```

"On the fly" initialization (Sec. 5.2):

get_variable

```
if   RUN
  & code(p) = get_variable(l,x_j,tt)
then
  mk_unbound(l,tt)
  bind(l,x_j)
  succeed
```

B.5. Unify instructions

Unify Variable

```
if   RUN
  & code(p) = unify_variable(l)
  & mode = Read        | mode = Write
then
  mk_unbound(l)        | mk_unbound(h)
  bind(l,nextarg)      | l ← <REF,h>
  nextarg := nextarg+  | h := h+
  succeed
```

Unify Value

```
if   RUN
  & code(p) = unify_value(l)
  & mode = Read        | mode = Write
then
  unify(l,nextarg)     | h ← l
  nextarg := nextarg+  | h := h+
  succeed
```

Unify Local Value

```
if   RUN
  & code(p) = unify_local_value(l)
  & mode = Read       |mode = Write
                      |& NOT(local(deref(l)))|local(deref(l))
then
  unify(l,nextarg)    |h ← deref(l)          |mk_heap_var(deref(l))
  nextarg := nextarg+ |h := h+               |
  succeed
```

"On the fly" initialization (Sec. 5.2):

unify_variable

```
if   RUN
  & code(p) = unify_variable(l,tt)
  & mode = Read       | mode = Write
then
  mk_unbound(l,tt)    | mk_unbound(h,tt)
  bind(l,nextarg)     | l ← <REF,h>
  nextarg := nextarg+ | h := h+
  succeed
```

B.6. Environment and Choicepoint Representation

The entries of the environment frame are stored in **STACK** at fixed offsets from the environment pointer `e` (ignoring cut points at this stage, but see 5.3). In particular, the environment also contains the variables $y_1, \ldots, y_n$ where n is the second parameter of the last call being executed (which is accessible via `cp-`):

```
ce(l)           ≡ l + 1
cp'(l)          ≡ l + 2
yi              ≡ e + 2 + i    (1≤i≤stack_offset(cp))
yi(l)           ≡ l + 2 + i    (1≤i≤stack_offset(val(cp'(l))))
stack_offset(l) ≡ n            if code(l-) = call(g,a,n)
tos(b,e)        ≡ if b ≤ e
                     then e + 2 + stack_offset(cp)
                     else b
```

Similarly, the choicepoint information is stored in **STACK** at fixed offsets from the backtracking pointer `b`. The choicpoint also contains the argument registers $x_1, \ldots, x_i$ of the current goal:

```
h(l)   ≡  l                 cp(l)   ≡  l - 4
tr(l)  ≡  l - 1             e(l)    ≡  l - 5
p(l)   ≡  l - 2             xi      ≡  l - 5 - i
b(l)   ≡  l - 3             hb(l)   ≡  val(h(b))
```

B.7. Indexing and Switching

try_me_else/try

```
if   RUN
  & code(p) =
        try_me_else(N,n) | try(L,n)
then
  let new_b = tos(b,e)+n+6
  b := new_b
  val(b(new_b)) := b
  store_state_in(new_b,n)
  val(p(new_b)) := N       | val(p(new_b)) := p+
  p := p+                  | p := L
```

retry_me_else/retry

```
if   RUN
  & code(p) =
          retry_me_else(N,n) | retry(L,n)
then
  fetch_state_from(b,n)
  val(p(b)) := N             | val(p(b)) := p+
  p:= p+                     | p := L
```

trust_me/trust

```
if   RUN
  & code(p) =
          trust_me(n) | trust(L,n)
then
  fetch_state_from(b,n)
  b := val(b(b))
  p := p+              | p := L
```

switch_on_term

```
if  RUN
  & code(p) = switch_on_term(i,Lv,Ls)
  & tag(deref(x_i)) = VAR | tag(deref(x_i)) = STRUC
then
    p := Lv              | p := Ls
```

switch_on_structure

```
if  RUN
  & code(p) = switch_on_structure(i,T)
then
  p := select(T,val(ref(deref(x_i))))
```

Abbreviations:

```
store_state_in(t,n) ≡
     FORALL i = 1,...,n
       val(x_i(t) := x_i
     ENDFORALL
     val(e(t))  := e
     val(cp(t)) := cp
     val(tr(t)) := tr
     val(h(t))  := h
```

```
fetch_state_from(t,n) ≡
     FORALL i = 1,...,n
       x_i := val(x_i(t))
     ENDFORALL
     e  := val(e(t))
     cp := val(cp(t))
     tr := val(tr(t))
     h  := val(h(t))
```