[**To appear in:** *Formal Aspects of Computing*, Springer-Verlag.]

Specification and correctness proof of a WAM extension for type-constraint logic programming

Christoph Beierle
Fachbereich Informatik
FernUniversität Hagen
Bahnhofstr. 48
D-58084 Hagen, Germany
beierle@fernuni-hagen.de

Egon Börger
Dipartimento di Informatica
Università di Pisa
Corso Italia 40
I-56100 Pisa, Italia
boerger@di.unipi.it

February 11, 1993; revised July 10, 1995

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 the first part of the paper, we keep the notion of types and dynamic type constraints rather 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.

In the second part of the paper, we refine the type constraints to the polymorphic order-sorted types of PROTOS-L. This allows us to develop a detailed, yet due to the use of evolving algebras, mathematically precise account of the PAM's compiled type constraint representation and solving facilities, and to extend the correctness theorem to compilation on the fully specified PAM.

Acknowledgements: 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.

Contents

1 Introduction

Recently, Gurevich's evolving algebra approach ([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 paper 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]) 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.

In the first part of this paper, we keep the notion of types and dynamic type constraints rather 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 Part I 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 Part I 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 algebras, we will already speak here in Part I in terms of PROTOS-L and PAM algebras (instead of type-constraint Prolog and type-constraint WAM algebras).

In Part II we 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, yet due to the use of the evolving algebras, 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 obtain as the final correctness theorem.

This paper was written in 1992/93 and revises and extends our work presented in [BB91] and [BB92]. It is organized as follows: Part I consists of Sections 2 - 5. 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 development carried out in Part I applies to any type system satisfying the given abstract conditions.

Part II consists of the Sections 6 - 8. Section 6 introduces the representation and constraint solving of monomorphic, order-sorted type constraints. Section 7 contains some type-specific optimizations of the PAM, which yields a situation where the WAM comes out as a special case of the PAM for any program not exploiting the advantages of dynamic type constraints. Section 8 gives a detailed account of polymorphic type constraint representation and solving in the PAM.

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

$\mathtt{f(t_1,\ldots,t_n)\ :=\ t}$

whose execution is to be understood as *changing* (or defining, if there was none) the value of function `f` at given arguments.

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

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

where *updates* may (and should) depend on the $\mathtt{t}_i$'s, setting the values of some functions on *newly created* elements $\mathtt{t}_i$ 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 $\mathtt{condition_1}$ | ... | $\mathtt{condition_n}$
then $\mathtt{updates_1}$ | ... | $\mathtt{updates_n}$

with pairwise incompatible conditions $\texttt{condition}_i$ 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` $\equiv$ `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:

$$\begin{tikzcd}
\mathcal{F}(B) \arrow[r, "\mathcal{F}(R)"] & \mathcal{F}(B') \\
B \arrow[u, "\mathcal{F}"] \arrow[r, "R"] & B' \arrow[u, "\mathcal{F}"']
\end{tikzcd}$$

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.

PART I: Adding type constraints to Prolog and the WAM

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

$$\texttt{subres:}\quad \mathbf{TERM} \times \mathbf{SUBST} \rightarrow \mathbf{TERM}$$

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

$$\mathbf{VARIABLE} \subseteq \mathbf{TERM}$$

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

$$\mathbf{EQUATION} \subseteq \mathbf{TERM} \times \mathbf{TERM}$$

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 $\mathtt{TOP} \in \mathbf{TYPETERM}$. Type constraints are given by the universe

$$\mathbf{TYPECONS} \subseteq \mathbf{TERM} \times \mathbf{TYPETERM}$$

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

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

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

$$\texttt{solution: } \mathbf{CSS} \rightarrow \mathbf{SUBST} \times \mathbf{TYPEPREFIX} \cup \{\texttt{nil}\}$$

where solution(CS) = nil iff solvable(CS) = false. For the trivially solvable empty constraint system we have

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

and the functions

$$\begin{array}{ll} \texttt{subst\_part:} & \mathbf{CSS} \rightarrow \mathbf{SUBST} \\ \texttt{prefix\_part:} & \mathbf{CSS} \rightarrow \mathbf{TYPEPREFIX} \end{array}$$

are the two obvious projections of solution. As an integrity constraint we assume

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

i.e., TOP is used to represent a trivially solvable type constraint.

These are the only assumptions we make about the universe **TYPETERM** until we introduce a special representation for it in Section 6. Thus, the complete development up to Section 5 (i.e. Part I of this paper) 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 t to be instantiated,
2. type constraints for the variables of t given by a prefix P_t, and
3. a constraint system CS to be applied.

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

$$\texttt{conres}(\texttt{t}, P_t, \texttt{CS}) = (\texttt{t}_1, P_1)$$

as a pair consisting of the instantiated term t_1 and type constraints P_1 for the variables of t_1. For this function

$$\begin{array}{ll} \texttt{conres:} & \mathbf{TERM} \times \mathbf{TYPEPREFIX} \times \mathbf{CSS} \rightarrow \\ & \mathbf{TERM} \times \mathbf{TYPEPREFIX} \cup \{\texttt{nil}\} \end{array}$$

we impose the following integrity constraints:

$$\begin{array}{l} \forall\ \texttt{t} \in \mathbf{TERM},\ P_t \in \mathbf{TYPEPREFIX},\ \texttt{CS} \in \mathbf{CSS}\ . \\ \textbf{if } \texttt{solvable}(P_t \cup \texttt{CS}) \textbf{ then} \\ \qquad\qquad \texttt{conres}(\texttt{t}, P_t, \texttt{CS}) = (\texttt{t}_1, P_1) \\ \qquad\qquad \textbf{where:} \\ \qquad\qquad \texttt{t}_1 = \texttt{subres}(\texttt{t}, \texttt{subst\_part}(\texttt{CS})) \\ \qquad\qquad P_1 = \texttt{prefix\_part}(P_t \cup \texttt{CS})_{|\texttt{var}(\texttt{t}_1)} \\ \qquad \textbf{else} \\ \qquad\qquad \texttt{conres}(\texttt{t}, P_t, \texttt{CS}) = \texttt{nil} \end{array}$$

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

$P\backslash X$ will be an abbreviation for $P_{|\texttt{domain}(P)\backslash\{X\}}$, 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

$$\mathbf{PROGRAM} \subseteq \mathbf{DEFCONTEXT} \times \mathbf{CLAUSE}^*$$

The definition context contains declarations of types, type constructors, etc. and will be refined in Part II. 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

$\{\texttt{P}\}$ `H <-- ` $\texttt{G}_1$ ` & ... & ` $\texttt{G}_n$`.`

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

$\{\texttt{P}\}$ $\texttt{G}_1$ ` & ... & ` $\texttt{G}_n$`.`

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

```
{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) |
    i ∈ NAT, H, G ∈ TERM, P ∈ TYPEPREFIX,
    N, L, Lv, Ls ∈ CODEAREA, T ∈ (ATOM × NAT × 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

`compile:` **CLAUSE** $\rightarrow$ **INSTR***

```
compile({P} H <-- G1 & ... & Gn) =
            [allocate, add_constraint(P), unify(H),
             call(G1),
             ...
             call(Gn),
             deallocate, proceed]
```

[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.

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

$$\begin{array}{ll} \texttt{+,-:} & \textbf{CODEAREA} \to \textbf{CODEAREA} \\ \texttt{code:} & \textbf{CODEAREA} \to \textbf{INSTR} \end{array}$$

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

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

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

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

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

The function

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

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.:

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

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

$$\texttt{chain:}\quad \textbf{CODEAREA} \to \textbf{CODEAREA}^*$$

we assume for an activator literal `act`

$$\texttt{chain(Ptr)} = \begin{cases} \texttt{chain(Lv)} & \text{if } \texttt{code(Ptr)} = \texttt{switch\_on\_term(i,Lv,Ls)} \\ & \quad \text{and } \texttt{is\_var(X}_i\texttt{)} \\ \texttt{chain(Ls)} & \text{if } \texttt{code(Ptr)} = \texttt{switch\_on\_term(i,Lv,Ls)} \\ & \quad \text{and } \texttt{is\_struct(X}_i\texttt{)} \\ \texttt{chain(select(T,f,a))} & \text{if } \texttt{code(Ptr)} = \texttt{switch\_on\_structure(i,T)} \\ & \quad \text{and } \texttt{f} = \texttt{functor(X}_i\texttt{)} \text{ and } \texttt{a} = \texttt{arity(X}_i\texttt{)} \\ \texttt{chain}_1\texttt{(Ptr)} & \text{otherwise} \end{cases}$$

$$\texttt{chain}_1\texttt{(Ptr)} = \begin{cases} \texttt{flatten[chain}_1\texttt{(Ptr),chain}_1\texttt{(N)]} & \text{if } \texttt{code(Ptr)} = \texttt{try\_me\_else(N)} \\ & \quad \text{or } \texttt{code(Ptr)} = \texttt{retry\_me\_else(N)} \\ \texttt{flatten[chain}_1\texttt{(C),chain}_1\texttt{(Ptr+)]} & \text{if } \texttt{code(Ptr)} = \texttt{try(C)} \\ & \quad \text{or } \texttt{code(Ptr)} = \texttt{retry(C)} \\ \texttt{chain}_1\texttt{(Ptr+)} & \text{if } \texttt{code(Ptr)} = \texttt{trust\_me} \\ \texttt{chain}_1\texttt{(C)} & \text{if } \texttt{code(Ptr)} = \texttt{trust(C)} \\ \texttt{[Ptr]} & \text{otherwise} \end{cases}$$

where $\texttt{X}_i = \texttt{arg(act,i)}$, `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 $\texttt{(f,a,pt)} \in \texttt{T}$.

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

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

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

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

STATE, **ENV** $\subseteq$ **STACK**
`-:` **STACK** $\rightarrow$ **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`	$\in$ **CSS**	`cs` $=$ $\emptyset$
`p`	$\in$ **CODEAREA**	`unload(p)` $=$ `[add_constraint(P),` `call(`G_1`),...,call(`G_n`),` `proceed]`
`cp`	$\in$ **CODEAREA**	`cp = p++`
`e`	$\in$ **ENV**	`vi'(e)=0, ct'(e)=nil, ce(e)=nil`
`b`	$\in$ **STATE**	`b = nil`
`vi`	$\in$ **NAT**	`vi = 0`
`ct`	$\in$ **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` $\equiv$ `subres(rename(G,vi'(e)),subst_part(cs))`

which is called the *current activator*.

The 0-ary function `prog` $\in$ **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` $\in$ $\{$`-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
  if solvable(cs ∪ {act ≐ rename(H,vi)})
  then cs := cs ∪ {act ≐ rename(H,vi)}
       vi := vi + 1
       succeed
  else backtrack
```

add_constraint

```
if   OK
  & code(p) = add_constraint(P)
then
  if solvable(cs ∪ rename(P,vi))
  then cs := cs ∪ rename(P,vi)
       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

$$(\mathbf{DATAAREA};\ \texttt{+},\ \texttt{-};\ \texttt{val})$$

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

$$\texttt{+},\ \texttt{-}:\quad \mathbf{DATAAREA} \rightarrow \mathbf{DATAAREA}$$

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

$$\texttt{val}:\quad \mathbf{DATAAREA} \rightarrow \mathbf{PO} + \mathbf{MEMORY} + \mathbf{SYMBOLTABLE}$$

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

$$\begin{array}{ll} \texttt{atom}: & \mathbf{SYMBOLTABLE} \rightarrow \mathbf{ATOM} \\ \texttt{arity}: & \mathbf{SYMBOLTABLE} \rightarrow \mathbf{NAT} \\ \texttt{entry}: & \mathbf{ATOM} \times \mathbf{NAT} \rightarrow \mathbf{SYMBOLTABLE} \end{array}$$

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

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

$$\begin{array}{ll} \texttt{tag}: & \mathbf{PO} \rightarrow \mathbf{TAGS} \\ \texttt{ref}: & \mathbf{PO} \rightarrow \mathbf{DATAAREA} + \mathbf{TYPETERM} \end{array}$$

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

$$\mathbf{TAGS} = \{\texttt{REF},\ \texttt{STRUC},\ \texttt{VAR}\}$$

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 6).

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

$$\begin{array}{ll} \texttt{tag(l)} & \equiv \texttt{tag(val(l))} \\ \texttt{ref(l)} & \equiv \texttt{ref(val(l))} \\ \texttt{l}_1 \leftarrow \texttt{l}_2 & \equiv \texttt{val(l}_1\texttt{)} := \texttt{val(l}_2\texttt{)} \end{array}$$

$$
\begin{array}{lll}
\texttt{l} \leftarrow \texttt{<t,r>} \equiv & \texttt{tag(l) := t} & \\
 & \texttt{ref(l) := r} & \\
\texttt{unbound(l)} \equiv & \texttt{tag(l) = VAR} & \\
\texttt{mk\_unbound(l)} & \equiv & \texttt{mk\_unbound(l,TOP)} \\
\texttt{mk\_unbound(l,tt)} & \equiv & \texttt{tag(l) := VAR} \\
 & & \texttt{insert\_type(l,tt)} \\
\texttt{insert\_type(l,tt)} & \equiv & \texttt{ref(l) := tt}
\end{array}
$$

where the last four abbreviations deal with the typed variable representation and where $\texttt{tt} \in$ **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

$$
\begin{array}{lll}
\texttt{deref:} & \mathbf{DATAAREA} \rightarrow & \mathbf{DATAAREA} \\
\texttt{term:} & \mathbf{DATAAREA} \rightarrow & \mathbf{TERM} \\
\texttt{type\_prefix:} & \mathbf{DATAAREA} \rightarrow & \mathbf{TYPEPREFIX}
\end{array}
$$

we assume for $\texttt{l} \in \mathbf{DATAAREA}$:

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

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

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

where $\texttt{make\_var(l)} \in \mathbf{VARIABLE}$ is a unique variable assigned to `l`. Note that the condition $\texttt{term(l)} \in \mathbf{TERM}$ now implies various consistency properties like:

$$
\begin{array}{lll}
\text{if } \texttt{unbound(l)} & \text{then} & \texttt{ref(l)} \in \mathbf{TYPETERM} \\
\text{if } \texttt{tag(l)} \in \{\texttt{REF, STRUC}\} & \text{then} & \texttt{ref(l)} \in \mathbf{DATAAREA} \\
 & & \texttt{term(l)} \in \mathbf{TERM} \\
 & & \texttt{typeprefix(l)} \in \mathbf{TYPEPREFIX} \\
\text{if } \texttt{tag(l) = STRUC} & \text{then} & \texttt{val(ref(l))} \in \mathbf{SYMBOLTABLE} \\
 & & \texttt{term(ref(l)+i)} \in \mathbf{TERM} \\
 & & \texttt{type\_prefix(ref(l)+i)} \in \mathbf{TYPEPREFIX}
\end{array}
$$

with $\texttt{i} \in \{\texttt{1,}\ldots\texttt{,arity(val(ref(l)))}\}$.

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` $\in$ **PDL** and

`ref':` **PDL**$\rightarrow$ **DATAAREA**

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

`left` $\equiv$ `ref'(pdl)` `right` $\equiv$ `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` $\in$ {`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` $\in$ **DATAARRA**, with `term` resp. `term'` values of `term(l)` and with `prefix` resp. `prefix'` values of `type_prefix(l)` before resp. after execution of `bind(`l_1`,`l_2`)`, we have if `unbound(`l_1`)` holds:

LET `CS = {mk_var(`l_1`) = term(`l_2`)}` $\cup$ `type_prefix(`l_1`)`
$\cup$ `type_prefix(`l_2`)`
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**

$$(\mathbf{HEAP};\ \mathtt{h},\ \mathtt{boh};\ \mathtt{+},\ \mathtt{-};\ \mathtt{val})$$

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

$$\mathbf{DATAAREA}\ +\ \mathbf{CODEAREA} \subseteq \mathbf{MEMORY}$$

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

$\mathtt{put\_value(y}_n\mathtt{,x}_j\mathtt{)}$
$\mathtt{put\_structure(f,x}_i\mathtt{)}$
$\mathtt{get\_value(y}_n\mathtt{,x}_j\mathtt{)}$
$\mathtt{get\_structure(f,x}_i\mathtt{)}$
$\mathtt{unify\_value(y}_n\mathtt{)}$
$\mathtt{unify\_variable(x}_n\mathtt{)}$
$\mathtt{put\_constraint(y}_n\mathtt{,tt)}$

with n,j,i $\in$ **NAT**, f $\in$ **SYMBOLTABLE**, tt $\in$ **TYPETERM**, $\mathtt{y}_n \in$ **DATAAREA**, $\mathtt{x}_i \equiv \mathtt{x(i)}$, where x: **NAT** $\rightarrow$ **AREGS** and **AREGS** $\subseteq$ **DATAAREA**. Note that $\mathtt{put\_constraint(y}_n\mathtt{,tt)}$ 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 $\mathtt{Y}_i$ of the term t to be built on the heap, there is already a term denoting $\mathtt{y}_i \in$ **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 $\mathtt{term(y}_i\mathtt{)}$ 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

$\mathtt{term(y}_n\mathtt{)} \in$ **TERM**
$\mathtt{type\_prefix(y}_n\mathtt{)} \in$ **TYPEPREFIX**

and X_i is a fresh variable, and CS is the constraint system consisting of the substitution associating every Y_n with $\mathtt{term(y}_n\mathtt{)}$ and of the union of the type constraints $\mathtt{type\_prefix(y}_n\mathtt{)}$, i.e.

$$\mathtt{CS} = \bigcup_n \{\mathtt{Y}_n \doteq \mathtt{term(y}_n\mathtt{)}\} \cup \mathtt{type\_prefix(y}_n\mathtt{)}$$

then the effect of setting p to

$$\mathtt{load(append(put\_code(X}_i\ \mathtt{=\ t),\ More))}$$

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

$$\mathtt{(term(x}_i\mathtt{),\ type\_prefix(x}_i\mathtt{))}$$

at the moment of passing to More, gets value of

$$\mathtt{conres(t,}\ \emptyset\mathtt{,\ CS)}$$

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 $\mathtt{x}_i$ in $\mathtt{get\_structure(f,x}_i\mathtt{)}$ is bound to a non-variable term (Appendix B.4). 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 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

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

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 $\mathtt{t} = \mathtt{f(\ldots)}$. Therefore, we introduce the function

$$\mathtt{propagate\_list:}\quad \mathbf{SYMBOLTABLE} \times \mathbf{TYPETERM} \rightarrow \mathbf{TYPETERM}^* \cup \{\mathtt{nil}\}$$

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 $\mathtt{tt}_j = \mathtt{TOP}$.

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`
  & $\mathtt{code(p)} = \mathtt{get\_structure(f,x}_i\mathtt{)}$

```
  & 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 l $\in$ **DATAAREA** and $\mathtt{tt}_1, \ldots, \mathtt{t}_n$ $\in$ **TYPETERM**, the update

```
mk_unbounds(l,(tt_1,...,tt_n)) ≡ FORALL i = 1,...,n DO
                                     mk_unbound(l+i,tt_i)
                                 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 $\{\mathtt{Y}_1, \ldots, \mathtt{Y}_l\}$, and if for n $\in \{1, \ldots, l\}$, $\mathtt{y}_n \in$ **DATAAREA** with

$\mathtt{unbound(y}_n\mathtt{)}$
$\mathtt{ref(y}_n\mathtt{)}$ $\in$ **TYPETERM**

and $\mathtt{X}_i$ is a fresh variable with $\mathtt{x}_i \in$ **DATAAREA** and

$\mathtt{term(x}_i\mathtt{)}$ $\in$ **TERM**
$\mathtt{type\_prefix(x}_i\mathtt{)}$ $\in$ **TYPEREFIX**

and CS is the constraint system consisting of the equation $\mathtt{t} \doteq \mathtt{term(x}_i\mathtt{)}$ together with $\mathtt{type\_prefix(x}_i\mathtt{)}$ and the union of the type constraints $\mathtt{type\_prefix(y}_n\mathtt{)}$, i.e.

$$\mathtt{CS} = \{\mathtt{t} \doteq \mathtt{term(x}_i\mathtt{)}\} \cup \mathtt{type\_prefix(x}_i\mathtt{)} \cup \bigcup_n \mathtt{type\_prefix(y}_n\mathtt{)}$$

then the effect of setting p to

$\mathtt{load(append(get\_code(X}_i\mathtt{ = t), More))}$

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`. ■

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

`put_constraint_seq(`$\{Y_1:tt_1, \ldots, Y_r:tt_r\}$`) = [put_constraint(`$y_1$`,`$tt_1$`),`
`...,`
`put_constraint(`y_r`,`tt_r`)]`

for which we use the new instruction `put_constraint(`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 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(`l_1`,tt)` we have if `unbound(`l_1`)` holds:

`(term', prefix') = conres(term, prefix`$\setminus\{$`mk_var(`l_1`)`$\}$`,` $\{$`mk_var(`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

$$\texttt{ref}^{\cdot\cdot}\texttt{:}\quad \textbf{TRAIL} \rightarrow \textbf{DATAAREA} \times \textbf{PO}$$

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` $\leq$ `l` $<$ `h & l` $<$ `hb  OR  bos` $\leq$ `l` $\leq$ `tos(b,e) & l` $<$ `b`.

For `t` $\in$ **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 $\texttt{l}_1$, $\texttt{l}_2$, `l` $\in$ **DATAAREA**. If `val(l)` before execution of $\texttt{bind(l}_1\texttt{,l}_2\texttt{)}$ is different from `val(l)` after successful execution of $\texttt{bind(l}_1\texttt{,l}_2\texttt{)}$, 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 $\texttt{l}_1$ and $\texttt{l}_2$ may be triggered by $\texttt{bind(l}_1\texttt{,l}_2\texttt{)}$; moreover, if e.g. $\texttt{l}_2$ denotes a polymorphic term containing variables these variables also have to be trailed if they get another type restriction in the binding process (see Sections 6.1 and 6.5).

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

$$\begin{array}{lll}
\texttt{call_seq(G)} & \mapsto & \texttt{call(G)} \\
\texttt{get_seq(H)} & \mapsto & \texttt{unify(H)} \\
\texttt{put_constraint_seq(P)} & \mapsto & \texttt{add_constraint(P)}
\end{array}$$

This correspondence also defines a (partial) function

$$\texttt{codepointer:}\quad \textbf{CODEAREA} \times \textbf{CODEAREA}_1$$

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

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

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 $\textbf{STATE}_1$ (resp. $\textbf{ENV}_1$) as a tuple of its `cs, p, cp, e, b` (resp. cp', ce) values, these functions are defined by:

$$\begin{array}{lll}
\texttt{term(l,l}_t\texttt{)} & & \text{yields the value } \texttt{term(l)} \text{ would take after having unwound} \\
& & \text{the trail down to } \texttt{l}_t \\
\texttt{typeprefix(l,l}_t\texttt{)} & & \text{yields the value } \texttt{type_prefix(l)} \text{ would take after having} \\
& & \text{unwound the trail down to } \texttt{l}_t \\
\texttt{css(l}_t\texttt{)} & = & \bigcup_{botr \le l < tr} \{\texttt{mk_var(location(l))} \doteq \texttt{term(location(l),l}_t\texttt{)}\} \\
& & \qquad \cup\ \texttt{typeprefix(location(l),l}_t\texttt{)} \\
\texttt{subst(l}_t\texttt{)} & = & \texttt{subst_part(css(l}_t\texttt{))} \\
\texttt{choicepoint(lb)} & = & \langle\ \texttt{css(val(tr(lb))),} \\
& & \quad \texttt{codepointer(val(p(lb))),} \\
& & \quad \texttt{codepointer(val(cp(lb))),} \\
& & \quad \texttt{env(val(e(lb))),} \\
& & \quad \texttt{choicepoint(val(b(lb)))}\ \rangle \\
\texttt{env(le)} & = & \langle\ \texttt{codepointer(val(cp(le))),} \\
& & \quad \texttt{env(val(ce(le)))}\ \rangle
\end{array}$$

The 0-ary functions are defined by

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

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

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

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

$$\begin{tikzcd}
\mathcal{F}(\mathtt{A}) \arrow[r, "\mathcal{F}(\mathtt{R})"] & \mathcal{F}(\mathtt{A'}) \\
\mathtt{A} \arrow[u, "\mathcal{F}"] \arrow[r, "\mathtt{R}"] & \mathtt{A'} \arrow[u, "\mathcal{F}"']
\end{tikzcd}$$

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(`y_n`,`x_j`)` instruction instead of `put_value(`y_n`,`x_j`)` for each unsafe occurrence of 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) $\leq$ e | deref(y_n) $>$ e
then
  x_j $\leftarrow$ deref(y_n) | mk_heap_var(deref(y_n))
                  | x_j $\leftarrow$ <REF,h>
  succeed

Note that the condition `deref(`y_n`)` $>$ `e` implies `unbound(deref(`y_n`))`. Thus, in case of y_n being unsafe, a new variable is created on the heap, referenced by both y_n and x_j. Unlike in Prolog, in PROTOS-L the type restriction of 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 will be introduced as an optimization in Section 7.) 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,x_j,tt)
then
  mk_unbound(l,tt)
  bind(l,x_j)
  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 $\mathtt{l} > \mathtt{e}$, the instruction $\mathtt{put\_variable(l,x}_j\mathtt{,tt)}$ ($\mathtt{get\_variable(l,x}_j\mathtt{,tt)}$, $\mathtt{unify\_variable(l,tt)}$, resp.) is equivalent to initializing $\mathtt{l}$ to unbound with $\mathtt{mk\_unbound(l)}$, executing $\mathtt{put\_constraint(l,tt)}$, and then executing $\mathtt{put\_unsafe\_value(l,x}_j\mathtt{)}$ ($\mathtt{get\_value(l,x}_j\mathtt{)}$, $\mathtt{unify\_local\_value(l)}$, resp.). For a permanent variable $\mathtt{Y}_n$, the instruction $\mathtt{put\_variable(y}_n\mathtt{,x}_j\mathtt{,tt)}$ is equivalent to initializing $\mathtt{y}_n$ to unbound with $\mathtt{mk\_unbound(y}_n\mathtt{)}$, executing $\mathtt{put\_constraint(y}_n\mathtt{,tt)}$, and then executing $\mathtt{put\_value(y}_n\mathtt{,x}_j\mathtt{)}$.

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 Section 7.2).

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 of Part I

Putting everything together developed so far, we obtain

Correctness Theorem 2 (Main Theorem of Part I): 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.

PART II: Polymorphic, order-sorted type constraints

6 PAM algebras with monomorphic type constraints

6.1 Binding

We are now ready for a first refinement of the binding update which will take into account the bind direction, occur check, and trailing, while the type constraints still remain abstract. We introduce two new 0-ary functions `arg1, arg2` $\in$ **DATAAREA** which will hold the locations given to the binding update, and extend the values of `what_to_do` by {`Bind_direction, Bind`} indicating that we have to choose the direction of the binding resp. do the binding itself. The new 0-ary function `return_from_bind` will take values of the domain of `what_to_do`, indicating where to return when the binding is finished. (Remember that the binding update is used in different places, e.g. in the unify update or in the creation of a new heap variable).

For $l_1, l_2 \in$ **DATAAREA** the binding update and some new abbreviations are defined by

```
bind(l1,l2)  ≡ arg1 := l1
               arg2 := l2
               return_from_bind := what_to_do
               what_to_do := Bind_direction
bind_success ≡ what_to_do := return_from_bind
BIND         ≡ OK & what_to_do = Bind
trail(l1,l2) ≡ ref¨(tr) := (l1, val(l1))
               ref¨(tr+):= (l2, val(l2))
               tr := tr++
```

In order to reset also the constant `what_to_do` upon backtracking, we refine the `backtrack` update to

```
backtrack ≡ p := val(p(b))
            unwind_trail
            what_to_do := Run
```

For `unbound(l1)` there are two alternative conditions on the update `occur_check(l1,l2)`, depending on whether the unification should perform the occur check (which is required for being logically correct) or not (which is done in most Prolog implementations for efficiency reasons):

OCCUR CHECK CONDITION: If no occur check should take place then the update `occur_check(l1,l2)` is empty; otherwise it has the following effect: If `mk_var(l1)` is among the variables of `term(l2)` then the `backtrack` update will be executed.

We will leave the occur check update abstract, and all correctness proofs are thus implicitly parameterized by the decision whether it actually performs the occur check or not.

Bind-1 (Bind-Direction)

```
if  OK
  & what_to_do = Bind_direction
  & unbound(arg1)
  & (NOT (unbound(arg2)) |  unbound(arg2)
      or                 | &
     arg2 < arg1)        |  arg2 > arg1      | arg1 = arg2
then
```

```
what_to_do := Bind      | what_to_do := Bind | bind_success
                        | arg1 := arg2       |
                        | arg2 := arg1       |
```

When binding two unbound variables their type constraints must be 'joined'. For this purpose we introduce the function

$$\texttt{inf:}\ \ \mathbf{TYPETERM} \times \mathbf{TYPETERM} \rightarrow \mathbf{TYPETERM}$$

which yields the *infimum* of two type terms, which may also be `BOTTOM` $\in$ **TYPETERM**. `TOP` and `BOTTOM` can be thought of as 'maximal' and 'minimal' type terms. As integrity constraints we have

$$\begin{aligned}
&\texttt{inf(TOP,tt) = inf(tt,TOP) = tt}\\
&\texttt{inf(BOTTOM,tt) = inf(tt,BOTTOM) = BOTTOM}\\
&\texttt{solution(}\{\texttt{t:BOTTOM}\}\texttt{) = nil}\\
&\texttt{solution(}\{\texttt{X:tt}_1\texttt{, X:tt}_2\}\texttt{) = solution(}\{\texttt{X:inf(tt}_1\texttt{,tt}_2\texttt{)}\}\texttt{)}
\end{aligned}$$

for any `t` $\in$ **TERM** and $\texttt{tt}_i \in$ **TYPETERM**.

Bind-2 (Bind-Var-Var)

```
if  BIND
  & unbound(arg2)
  & LET inf = inf(ref(arg1),ref(arg2))
  & inf ≠ BOTTOM                           | inf = BOTTOM
  & inf ≠ ref(arg2)      | inf = ref(arg2) |
then
  trail(arg1,arg2)       | trail(arg1)     | backtrack
  insert_type(arg2,inf)  |                 |
  arg1 ← <REF,arg2>                        |
  bind_success                             |
```

When binding an unbound variable to a non-variable term, the type restriction of the variable must be propagated to the variables occurring in the term. As a special case this situation already occured in $\texttt{get\_structure(f,x}_i\texttt{)}$ when the dereferenced value of $\texttt{x}_i$ is a type-restricted variable. In that situation where the term was still to be built upon the heap, we ensured the propagation by writing `arity(f)` free value cells on the heap with appropriate type restrictions and continuing in read mode; the actual propagation was then achieved by the immediately following sequence of `unify` instructions. In the general case occurring in the binding rules, the arguments of the term are not just variables but arbitrary terms. However, as we will not go into the details of type constraint solving here, we assume an abstract propagate update satisfying the following:

PROPAGATION CONDITION: For any $\texttt{l}_1\texttt{, l}_2\texttt{, l} \in$ **DATAARRA**, with `term` resp. `term'` values of `term(l)`, with `prefix` resp. `prefix'` values of `type_prefix(l)`, and with `val` resp. `val'` values of `val(l)`, before resp. after execution of $\texttt{propagate(l}_1\texttt{,l}_2\texttt{)}$ we have if $\texttt{unbound(l}_1\texttt{)}$, $\texttt{ref(l}_1\texttt{)} \in$ **TYPETERM**, $\texttt{tag(l}_2\texttt{) = STRUC}$, and $\texttt{term(l}_2\texttt{)} \in$ **TERM**:

LET $\texttt{CS = }\{\texttt{term(l}_2\texttt{):ref(l}_1\texttt{)}\}$
if `solvable(CS) = true`
then (a) `(term', prefix') = conres(term, prefix, CS)`
(b) if `val` $\neq$ `val'` then the location `l` will be trailed
else `backtrack` update will be executed

With this update at hand the third binding rule is

Bind-3 (Bind-Var-Struc)

```
if   BIND
  & NOT (unbound(arg2))
then
  trail(arg1)
  arg1 ← <REF,arg2>
  occur_check(arg1,arg2)
  propagate(arg1,arg2)
```

BINDING LEMMA 1: The bind rules are a correct realization of the binding update of Section 3.2, i.e. the BINDING CONDITIONS 1 and 3 (and thus also 2), the TRAILING CONDITION as well as the STACK VARIABLES PROPERTY are preserved.

Proof: The proof for the update $\texttt{bind(l}_1\texttt{,l}_2\texttt{)}$ is by case analysis and induction on the size of $\texttt{term(l}_2\texttt{)}$, relying on the integrity conditions for the infimum function on type terms when binding one type-restricted variable to another one (Bind-2), resp. on the Propagation Condition when binding a variable to a non-variable term (Bind-3). ■

6.2 Monomorphic, order-sorted types

Before introducing a representation for type terms we introduce some new functions and universes that are related to **TYPETERM**. Note that until now we have kept **TYPETERM** indeed abstract; it is only in this section that we come to some more specific type term characteristics such as monomorphic and polymorphic type terms. However, following our principle of stepwise refinement of the PAM development, we first deal only with monomorphic type constraints solving, while the details of polymorphic type constraint handling will still be kept abstract in this section.

On the universes **TYPETERM** and **SYMBOLTABLE** we introduce the functions

$$\begin{array}{ll} \texttt{is\_top:} & \textbf{TYPETERM} \rightarrow \textbf{BOOL} \\ \texttt{is\_monomorphic:} & \textbf{TYPETERM} \rightarrow \textbf{BOOL} \\ \texttt{is\_polymorphic:} & \textbf{TYPETERM} \rightarrow \textbf{BOOL} \end{array}$$

with their obvious meaning. The function

$$\texttt{target\_sort:} \quad \textbf{SYMBOLTABLE} \rightarrow \textbf{SORT}$$

yields the target sort of a constructor, where **SORT** is a new universe, representing sort names. It comes with a function

$$\texttt{subsort:} \quad \textbf{SORT} \times \textbf{SORT} \rightarrow \textbf{BOOL}$$

defining the order relation on the monomorphic sorts (and being undefined on the polymorhic sorts [Bei90]), respectively. For the refinement of type constraint handling we assume two functions

$$\begin{array}{ll} \texttt{sort\_glb:} & \textbf{SORT} \times \textbf{SORT} \rightarrow \textbf{SORT} \\ \texttt{poly\_inf:} & \textbf{TYPETERM} \times \textbf{TYPETERM} \rightarrow \textbf{TYPETERM} \end{array}$$

that refine the `inf` function (from 6.1) in the sense that for any $\texttt{tt}_1$, $\texttt{tt}_2 \in \textbf{TYPETERM}$

$$\texttt{inf(tt}_1\texttt{,tt}_2\texttt{)} = \begin{cases} \texttt{sort\_glb(tt}_1\texttt{,tt}_2\texttt{)} & \text{if } \texttt{is\_monomorphic(tt}_1\texttt{)} \\ & \quad \text{and } \texttt{is\_monomorphic(tt}_2\texttt{)} \\ \texttt{poly\_inf(tt}_1\texttt{,tt}_2\texttt{)} & \text{if } \texttt{is\_polymorphic(tt}_1\texttt{)} \\ & \quad \text{and } \texttt{is\_polymorphic(tt}_2\texttt{)} \end{cases}$$

For constraint solving involving a monomorphic type term `s` and `t = f(...)` $\in$ **TERM** we have the integrity constraint

$$\texttt{solution}(\{\texttt{t}:\texttt{s}\}) = \begin{cases} \emptyset & \text{if } \texttt{subsort(target\_sort(f),s)} \\ \texttt{nil} & \text{otherwise} \end{cases}$$

i.e. the solvability of a monomorphic type constraint depends solely on the subsort relationship between the required sort and the target sort of the top-level constructor of the term. It will turn out that this suffices for the refinement of monomorphic type constraint handling.

6.3 Representation of types

For the PAM representation of typeterms we introduce a pointer algebra, similar to DATAAREA, which will be used for the representation of both monomorphic types and polymorphic type terms (for the latter see Section 8):

(**TYPEAREA**; `ttop, tbottom, TOP; +, -; tval`)

`ttop, tbottom, TOP:` $\rightarrow$ **TYPEAREA**
`+, -:` **TYPEAREA** $\rightarrow$ **TYPEAREA**
`tval:` **TYPEAREA** $\rightarrow$ **TO**

The functions `ttag` and `tref` are defined on the universe of "type objects" **TO**

`ttag:` **TO** $\rightarrow$ **TTAGS**
`tref:` **TO** $\rightarrow$ **SORT** + **TYPEAREA**

with the tags for type terms given by (to be extended later)

{ `S_TOP, S_MONO, S_POLY` } $\subseteq$ **TTAGS**

Similar as done before, we abbreviate `ttag(tval(l))` and `tref(tval(l))` by `ttag(l)` and `tref(l)`. As integrity constraints we have

if `ttag(l) = S_MONO`	then	`tref(l)` $\in$ **SORT**
		`is_monomorphic(tref(l))`
if `ttag(l) = S_POLY`	then	`is_polymorphic(tref(l))`

where the auxiliary function

`typeterm:` **TYPEAREA** $\rightarrow$ **TYPETERM**

satisfies the constraints

`typeterm(l) = TOP`	if `ttag(l) = S_TOP`
`typeterm(l) = tref(l)`	if `ttag(l) = S_MONO`

We refine the PAM algebras of Section 5 by replacing the universe **TYPETERM** by its representing universe **TYPEAREA**. The codomain of the `ref` function (from 3.1) now contains **TYPEAREA**, and in the integrity constraints of 3.1 as well as in the definition of `type_prefix` the case for `unbound(l)` now contains `typeterm(ref(l))` instead of `ref(l)`. The three abstract functions `is_top`, `is_monomorphic`, and `is_polymorphic` defined on **TYPETERM** are defined on **TYPEAREA** by just looking at the type tag; for `l` $\in$ **DATAAREA** we therefore use the following abbreviations:

`top(l)`	$\equiv$	`tag(l) = VAR & ttag(ref(l)) = S_TOP`	
`monomorphic(l)`	$\equiv$	`tag(l) = VAR & ttag(ref(l)) = S_MONO`	
`polymorphic(l)`	$\equiv$	`tag(l) = VAR & ttag(ref(l)) = S_POLY`	
`sort(l)`	$\equiv$	`typeterm(ref(l))`	if `monomorphic(l)`

6.4 Initialization of type constrained variables

In the PAM algebras developed so far the update `insert_type(l,t)` is used - as part of the `mk_unbound` update - in the variable initialization instructions `get_variable`, `put_variable`, and `unify_variable` (Section 5.2). (Its use in the multiple `mk_unbounds` update in `get_structure` will be refined in Section 6.6 below). This update is now refined by

```
insert_type(l,tt) ≡ if is_top(tt)
                      then insert_top(l)
                      else if is_monomorphic(tt)
                           then insert_mono(l,tt)
                           else insert_poly(l,tt)
insert_top(l)     ≡ ref(l) := ttop
                    ttag(ttop) := S_TOP
                    ttop := ttop+
insert_mono(l,s)  ≡ ref(l) := ttop
                    ttag(ttop) := S_MONO
                    tref(ttop) := s
                    ttop := ttop+
```

where we use a new type area location when inserting a monomorphic sort `s` (resp. `TOP`) as restriction for location `l` $\in$ **DATAAREA**. [2]

Similarly, the insertion of polymorphic type terms by `insert_poly(l,tt)` will be handled in Section 8. As we want to leave the details of polymorphic type constraint solving still abstract here, we pose the following

POLYMORPHIC TYPE INSERTION CONDITION: For any $\mathtt{l}_1$, `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_poly(`$\mathtt{l}_1$`,tt)`, we have if `unbound(`$\mathtt{l}_1$`)` and `tt` $\in$ **TYPETERM** with `is_polymorphic(tt)`:

$$(\mathtt{term'},\ \mathtt{prefix'}) = \mathtt{conres(term,\ prefix}\backslash\mathtt{mk\_var(l_1),\ \{mk\_var(l_1):tt\})}$$

TYPE INSERTION LEMMA: The refinement of the `insert_type` update satisfies the TYPE INSERTING CONDITION of 3.5.

Proof: By straightforward case analysis for `TOP`, monomorphic and polymorphic type restrictions; for the latter the POLYMORPHIC TYPE INSERTION CONDITION is used. ■

6.5 Binding of type constrained variables

We refine the binding rules of Section 6.1 according to the type term representation. Rule Bind-1 remains unchanged, whereas the rule Bind-2 for binding two variables is replaced by the following four rules:

Bind-2a (Bind-TOP-Any)

if `BIND`
  `& top(arg1)`
  `& unbound(arg2)) | NOT (unbound(arg2))`

[2] Note that deliberately we have left out the re-use of type area locations. For trailing, we have to preserve old type restrictions to be recovered upon backtracking. However, locations that will not be reached any more by backtracking can be re-used, just as e.g. memory on the local stack or on the heap is freed for re-use upon backtracking. In the current PAM implementation the type area is embedded into the heap so that the same mechanism for allocating and deallocating can be used. However, other realizations are also possible, and we will not elaborate this topic in this paper.

```
then
  trail(arg1)
  arg1 ← <REF,arg2>
  bind_success      | occur_check(arg1,arg2)
```

Bind-2b (Bind-Var-TOP)

```
if  BIND
  & monomorphic(arg1) OR polymorphic(arg1)
  & top(arg2)
then
  trail(arg1,arg2)
  arg1 ← <REF,arg2>
  arg2 ← arg1
  bind_success
```

Bind-2c (Bind-Mono-Mono)

```
if  BIND
  & monomorphic(arg1)
  & monomorphic(arg2)
  & LET glb = sort_glb(sort(arg1),sort(arg2))
  & glb ≠ BOTTOM                             | glb = BOTTOM
  & glb ≠ sort(arg2)    | glb = sort(arg2) |
then
  trail(arg1,arg2)      | trail(arg1)      | backtrack
  insert_type(arg2,glb) |                  |
  arg1 ← <REF,arg2>                        |
  bind_success                             |
```

Bind-2d (Bind-Poly-Poly)

```
if  BIND
  & polymorphic(arg1)
  & polymorphic(arg2)
then
  trail(arg1)
  arg1 ← <REF,arg2>
  poly_infimum(arg1,arg2)
```

The only still abstract update in these rules is the poly_infimum(l_1,l_2) update used when binding two polymorphically restricted variables, for which we require the following

POLYMORPHIC INFIMUM CONDITION: For any l_1, l_2, l $\in$ **DATAAREA**, with term resp. term' values of term(l), with prefix resp. prefix' values of type_prefix(l), and with val resp. val' values of val(l), before resp. after execution of poly_infimum(l_1,l_2) we have if for i = 1,2 unbound(l_i), polymorphic(l_i), and typeterm(ref(l_i)) $\in$ **TYPETERM**:

> LET CS = {mk_var(l_2):poly_inf(typeterm(l_1),typeterm(l_2))}
> if solvable(CS) = true
> then (a) (term', prefix') = conres(term, prefix, CS)
> (b) if val $\neq$ val' then the location l will be trailed
> else backtrack update will be executed

Rule Bind-3 of Section 6.1 for binding a variable to a non-variable structure is replaced by the rules Bind-2a above (which already covers the case that the variable has no type restriction, denoted by `TOP`) and the two new rules

Bind-3a (Bind-Mono-Struc)

```
if   BIND
   & monomorphic(arg1)
   & NOT (unbound(arg2))
   & subsort(target_sort(ref(arg2)),sort(arg1))
       = true            |     = false
then
   trail(arg1)           | backtrack
   arg1 ← <REF,arg2>     |
   occur_check(arg1,arg2) |
```

Bind-3b (Bind-Poly-Struc)

```
if   BIND
   & polymorphic(arg1)
   & NOT (unbound(arg2))
then
   trail(arg1)
   arg1 ← <REF,arg2>
   occur_check(arg1,arg2)
   poly_propagate(arg1,arg2)
```

The abstract update `poly_propagate(`l_1`,`l_2`)` must satisfy the

POLYMORPHIC PROPAGATION CONDITION which is obtained from the PROPAGATION CONDITION of 6.1 by adding `is_polymorphic(`l_1`)` as an additional precondition and replacing `ref(`l_1`)` by `typeterm(ref(`l_1`))`.

BINDING LEMMA 2: The refined binding rules are a correct realization of the binding rules of Section 6.1 and thus also of the binding update of 3.2.

Proof: Following the proof of the BINDING LEMMA in 6.1 we have to show that the rules Bind-2a - Bind-2d and Bind-3a - Bind-3b are correct realizations of the `inf` function used in Bind-2 and of the `propagate` update used in Bind-3. This follows by straightforward case analysis for `TOP`, monomorphic, and polymorphic type restrictions: For `TOP`, we use its property that it is 'maximal' w.r.t. `inf` and that the `propagate` update can not have any effect since any `TOP` restriction trivially holds (Section 2.1). For the monomorphic case we conclude from the last integrity constraint given in Section 6.2 that the `propagate` update is either empty or fails immediately due to the subsort test, implying that the different cases correctly simulate this situation. For the polymorphic case the POLYMORPHIC INFIMUM and POLYMORPHIC PROPAGATION CONDITIONS are used. ■

6.6 Getting of structures

We refine the `get_struture` rules of Section 3.4 according to the type term representation. Rule Get-Structure-1 remains unchanged, whereas the rule Get-Structure-2 for the case that $\mathbf{x}_i$ is an unbound variable is replaced by the following two rules:

Get-Structure-2a

```
if   RUN
   & code(p) = get_structure(f,xi)
   & monomorphic(deref(xi))
   & NOT ( subsort(target_sort(f),sort(deref(xi))) )
then
   backtrack
```

Get-Structure-2b

```
if   RUN
   & code(p) = get_structure(f,xi)
   & top(deref(xi))            | polymorphic(deref(xi))
          OR                   |
     (monomorphic(deref(xi)) & |
      subsort(target_sort(f),  |
              sort(deref(xi))) |
then
   h ← <STRUC,h+>
   bind(deref(xi),h)
   val(h+) := f
   h := h++                    | h := h + arity(f) + 2
   mode := Write               | nextarg := h++
                               | mode := Read
                               | FORALL i = 1,...,arity(f) DO
                               |    mk_unbound(h+i)
                               | ENDFORALL
                               | poly_propagate(h+,deref(xi))
   succeed
```

Thus, the only remaining abstract update is in the case when $\mathtt{x}_i$ is a polymorphically restricted variable; this case in Get-Structure-2b is reduced to the more general update `poly_propagate` already introduced in the previous subsection.

CORRECTNESS OF GET-STRUCTURE REFINEMENT: The refined Get-Structure rules are a correct realization of the rules of Section 3.4, i.e. the GETTING LEMMA stills holds for the refined type term representation.

Proof: As in the proof of the BINDING LEMMA 2 in the previous subsection we can apply a straightforward case analysis for `TOP`, monomorphic, and polymorphic type restrictions: For `TOP`, we observe that always both conditions `can_propagate(f,TOP)` and `trivially_propagates(f,TOP)` used in the Get-Structure rule of 3.4 hold. For monomorphic type restrictions, the propagation reduces again to the subsort test. For the polymorphic case the POLYMORPHIC PROPAGATION CONDITION ensures that exactly the type restrictions given by the `propagate_list` function used in 3.4 are propagated onto the arguments of the structure. ■

Whereas we have now introduced a representation for type terms and rules for monomorphic type constraint solving, some details of polymorphic type constraint solving are still abstract, namely the three updates `insert_poly(l,tt)`, `poly_infimum(`$\mathtt{l}_1$`,`$\mathtt{l}_2$`)`, and `poly_propagate(`$\mathtt{l}_1$`,`$\mathtt{l}_2$`)` which will be refined in Section 8.

7 PAM Optimizations

7.1 Special representation for typed variables

Many of the type related rules introduced above - in particular the get-structure and the binding rules - apply only if the involved variable has no type restriction at all (i.e. `TOP`), or a monomorphic, or a polymorphic type restriction, respectively. In the spirit of the WAM's tagged architecture it is thus sensible to distinguish these three different cases efficiently by special tags [BMS91]. The tag `VAR` is therefore replaced by the three tags `FREE`, `FREE_M`, `FREE_P`.

Moreover, in the representation of monomorphic sorts one can also easily save a type area location by letting the `ref` value of a data area location point directly to **SORT**. Therefore, we extend the codomain of the function `ref` (see 3.1) to include also **SORT**. Let `l` $\in$ **DATAAREA**; instead of

```
val(l) = <VAR,t>      and      tval(t) = <S_MONO,s>
```

we will just have

```
val(l) = <FREE_M,s>
```

and instead of

```
val(l) = <VAR,t>      and      ttag(t) = S_TOP
```

we will just have

```
tag(l) = FREE
```

This motivates the following modified abbreviations:

```
mk_unbound(l)          ≡ tag(l) := FREE
mk_unbound_mono(l,s)   ≡ tag(l) := FREE_M
                         ref(l) := s
mk_unbound_poly(l,tt)  ≡ tag(l) := FREE_P
                         insert_poly(l,tt)
mk_unbound(l,tt)       ≡ if is_top(tt)
                            then mk_unbound(l)
                            elseif is_monomorphic(tt)
                                 then mk_unbound_mono(l,tt)
                                 else mk_unbound_poly(l,tt)
unbound(l)        ≡ tag(l) ∈ {FREE, FREE_M, FREE_P}
top(l)            ≡ tag(l) = FREE
monomorphic(l)    ≡ tag(l) = FREE_M
polymorphic(l)    ≡ tag(l) = FREE_P
sort(l)           ≡ ref(l)                     if monomorphic(l)
```

The integrity constraint for the case `unbound(l)` of Section 3.1 is replaced by

```
if tag(l) = FREE_M   then  ref(l) ∈ SORT
if tag(l) = FREE_P   then  ref(l) ∈ TYPEAREA
                           typeterm(ref(l)) ∈ TYPETERM
                           is_polymorphic(typeterm(ref(l)))
```

and in the definition of `type_prefix` the case for `unbound(l)` is refined to

$$\texttt{type\_prefix(l)} = \begin{cases} \texttt{mk\_var(l)}:\texttt{TOP} & \text{if } \texttt{tag(l) = FREE} \\ \texttt{mk\_var(l)}:\texttt{ref(l)} & \text{if } \texttt{tag(l) = FREE\_M} \\ \texttt{mk\_var(l)}:\texttt{typeterm(ref(l))} & \text{if } \texttt{tag(l) = FREE\_P} \\ \ldots & \end{cases}$$

Every time a new variable is created, this refined representation of variables will be taken into account by one of the specialized `mk_unbound` updates introduced above; for instance in the Get-Structure-2b rule (Section 6.6).

Similarly, the rules for initializing variables (Section 5.2) are modified as explained in the following. In order to take advantage of the refined variable representation we modify the compile function such that each instruction of the form

```
get_variable(l,x_j,tt)
```

is replaced by one of the three new instructions

```
get_free(l,x_j)
get_mono(l,x_j,tt)
get_poly(l,x_j,tt)
```

depending on whether `is_top(tt)`, `is_monomorphic(tt)`, or `is_polymorphic(tt)` holds. Likewise, all `put_variable` and `unify_variable` instructions are replaced by the instructions

```
put_free(l,x_j)          unify_free(l)
put_mono(l,x_j,tt)       unify_mono(l,tt)
put_poly(l,x_j,tt)       unify_poly(l,tt)
```

respectively. Note that these new instructions always correspond to the *first* occurrence of a variable in a clause and are thus responsible for the correct type initialization of that variable.

Put-1 (X variable)

```
if   RUN
  & code(p) =
      put_free(x_i,x_j) | put_mono(x_i,x_j,s)   | put_poly(x_i,x_j,tt)
then
  mk_unbound(h)       | mk_unbound_mono(h,s) | mk_unbound_poly(h,tt)
  x_i ← <REF,h>
  x_j ← <REF,h>
  h := h+
  succeed
```

Put-2 (Y variable)

```
if   RUN
  & code(p) =
      put_free(y_n,x_j) | put_mono(y_n,x_j,s)   | put_poly(y_n,x_j,tt)
then
  mk_unbound(y_n)     | mk_unbound_mono(y_n,s)| mk_unbound_poly(y_n,tt)
  x_j ← <REF,y_n>
  succeed
```

Get (Variable)

```
if   RUN
  & code(p) =
      get_free(l,x_j) | get_mono(l,x_j,s)    | get_poly(l,x_j,tt)
then
  l ← x_j             | mk_unbound_mono(l,s)| mk_unbound_poly(l,tt)
                      | bind(l,x_j)          | bind(l,x_j)
  succeed
```

Unify (Read Mode)

```
if   RUN
   & code(p) =
       unify_free(l)  | unify_mono(l,s)     | unify_poly(l,tt)
   & mode = Read
then
   l ← <REF,nextarg> | mk_unbound_mono(l,s)| mk_unbound_poly(l,tt)
                     | bind(l,nextarg)     | bind(l,nextarg)
   nextarg := nextarg+
   succeed
```

Unify (Write Mode)

```
if   RUN
   & code(p) =
       unify_free(l) | unify_mono(l,s)     | unify_poly(l,tt)
   & mode = Write
then
   mk_unbound(h)     | mk_unbound_mono(h,s)| mk_unbound_poly(h,tt)
   l ← <REF,h>
   h := h+
   succeed
```

CORRECTNESS OF REFINED VARIABLE REPRESENTATION: The PAM algebras with the refined variable representation are correct with respect to the PAM algebras constructed in Section 6.

Proof: The only type inserting update of 6.4 that is still used is `insert_poly` for which the POLYMORPHIC TYPE INSERTION CONDITION ensures the TYPE INSERTION CONDITION. Inserting `TOP` and monomorphic type restrictions for variables obviously has the same effect as in 6.4. Trailing still works fine since in 4.2 we trailed the complete `val` decoration of a data area location - including its tag - and restored it upon backtracking. With these two main observations the proof follows by case analysis for the three different kinds of type restrictions. Showing that each variable is initialized properly is straightforward, and the correct treatment of the thus refined variable representation in all relevant rules (in particular the binding rules) is ensured directly by our modified abbreviations that refer to a variable's representation, like `monomorphic(l)` or `sort(l)`. ∎

7.2 Switch on Types

As opposed to the WAM, in the PAM also a switch on the subtype restriction of a variable is possible (c.f. 5.3) which increases the determinancy detection abilities. Since only monomorphic types can have explicitly defined subtypes there are two switch-on-term instructions. (Note that in this paper we did not introduce special representations for constants, lists, or built-in integers; they are, however, present in the PAM and could be added to our treatment without difficulties, which would lead to additional parameters in the following instructions.)

Switch-on-poly-term

```
if   RUN
   & code(p) = switch_on_poly_term(i,Lfree,Lstruc)
   & tag(deref(x_i)) ∈ {FREE, FREE_P} | tag(deref(x_i)) = STRUC
then
   p := Lfree                         | p := Lstruc
```

The `switch_on_poly_term` instruction is as the WAM `switch_on_term` instruction (c.f. B.7) except that the variable may carry a polymorphic type restriction, which however does not lead to the exclusion of any clauses, since in PROTOS-L no explicit subtype relationships are allowed between polymorphic types [Bei90].

Switch-on-mono-term

```
if   RUN
  & code(p) = switch_on_mono_term(i,Lfree,Lfree_m,Lstruc)
  & tag(deref(x_i)) =
       FREE    | FREE_M        | STRUC
then
  p := Lfree   | p := Lfree_m  | p := Lstruc
```

In the `switch_on_mono_term` instruction we distinguish the two cases for a `FREE` variable and a `FREE_M` variable. In the first case again no clauses can be excluded form further consideration, but in the second case only those clauses that are compatible with x_i's subtype restriction have to be taken into account. The latter is achieved by setting the program counter `p` to a label where a `switch_on_sort` instruction will exploit x_i's subtype restriction:

Switch-on-sort

```
if   RUN
  & code(p) = switch_on_sort(i,Table)
then
  p := select_sort(Table,sort(deref(x_i)))
```

where `Table` is a list of pairs of the form **SORT** $\times$ **CODEAREA**, and $\texttt{select}_{sort}$`(Table,s)` yields the location `c` such that `(s,c)` is in `Table`.

In order to establish a correctness proof for the extended switching instructions we must extend the assumptions on the compiler stated in 2.2. The defintion of `chain` is changed so that the two cases for `switch_on_term` are replaced by

$$
\texttt{chain(Ptr)} \quad = \quad \begin{cases}
\texttt{chain(Lf)} & \text{if } \texttt{code(Ptr)} = \texttt{switch\_on\_poly\_term(i,Lf,Ls)} \\
 & \quad \text{and } \texttt{is\_top(X}_i\texttt{)} \text{ or } \texttt{is\_polymorphic(X}_i\texttt{)} \\
\texttt{chain(Ls)} & \text{if } \texttt{code(Ptr)} = \texttt{switch\_on\_poly\_term(i,Lf,Ls)} \\
 & \quad \text{and } \texttt{is\_struct(X}_i\texttt{)} \\
\texttt{chain(Lf)} & \text{if } \texttt{code(Ptr)} = \texttt{switch\_on\_mono\_term(i,Lf,Lfm,Ls)} \\
 & \quad \text{and } \texttt{is\_top(X}_i\texttt{)} \\
\texttt{chain(Lfm)} & \text{if } \texttt{code(Ptr)} = \texttt{switch\_on\_mono\_term(i,Lf,Lfm,Ls)} \\
 & \quad \text{and } \texttt{is\_monomorphic(X}_i\texttt{)} \\
\texttt{chain(Ls)} & \text{if } \texttt{code(Ptr)} = \texttt{switch\_on\_mono\_term(i,Lf,Lfm,Ls)} \\
 & \quad \text{and } \texttt{is\_struct(X}_i\texttt{)} \\
\texttt{chain(select}_{sort}\texttt{(T,s))} & \text{if } \texttt{code(Ptr)} = \texttt{switch\_on\_sort(i,T)} \\
 & \quad \text{and } \texttt{s} = \texttt{sort(X}_i\texttt{)} \\
\ldots &
\end{cases}
$$

SWITCHING LEMMA: Switching extended to types preserves correctness.

Proof: By case analysis using the extended `chain` definition, and relying on the correctness of the other building blocks of the determinancy detection mechanism (like `try, retry, trust`, etc.) which remain unchanged. ■

Note that the special representation of typed variables introduced in this section lead to the situation that the type extension in the PAM is indeed orthogonal to the WAM. Any untyped program is carried out in the PAM with the same efficiency as in the WAM: Adding the trivial one-sorted type information to such a program reveals that the PAM code will contain only the FREE-case for variables. Apart form the minor difference of representing a free (unconstrained) variable not by a selfreference (as in the WAM) but by a special tag, the generated and executed code is thus exactly the same for both the WAM and the PAM. On the other hand, any typed program exploiting e.g. the possibilities of computing with subtypes can take advantage of the type constraint handling facilities in the PAM which would have to be simulated by additional explicit program clauses in an untyped version.

8 Polymorphic type constraint solving

In this section polymorphic type constraint handling is refined by refining the three updates `insert_poly(l,tt)`, `poly_infimum(`l_1`,`l_2`)`, and `poly_propagate(`l_1`,`l_2`)` that have been left abstract so far.

8.1 Representation of polymorphic type terms

For the representation of polymorphic type terms we introduce an additional function on **SORT**

`sort_arity:` **SORT** $\rightarrow$ **NAT**

yielding the arity of a polymorphic sort (which must be 0 in the case of a monomorphic sort). The relationship between the declaration part of the program `prog` (see 2.1 and 2.4) and the functions on **SORT** is regulated by the following integrity constraints: For each function declaration of the form

`f:` $d_1 \ldots d_m \rightarrow$ `s(`α_1`,...,`α_n`)`

with `m, n` $\geq$ `0`, pairwise distinct (type) variables α_i that occur in $d_1,\ldots,d_m$, and each `tt = s(...)` $\in$ **TYPETERM** the following holds:

```
target_sort(entry(f, m)) = s
arity(entry(f, m)) = m
sort_arity(s) = n
is_monomorphic(tt) = true    iff    n = 0
is_polymorphic(tt) = true    iff    n > 0
```

Let use illustrate these integrity constraints by some examples. Consider the three function declarations

```
succ:    nat → nat
cons:    α × list(α) → list(α)
mk_pair: α × β → pair(α,β)
```

Then we have e.g. the following relationships:

```
target_sort(entry(succ,1))    = nat
target_sort(entry(cons,2))    = list
target_sort(entry(mk_pair,2)) = pair

arity(entry(succ,1))   = 1
arity(entry(cons,2))   = 2
arity(entry(mk_pair,2) = 2

sort_arity(nat)  = 0
sort_arity(list) = 1
sort_arity(pair) = 2

is_monomorphic(nat)           = true
is_polymorphic(list(list(γ))) = true
```

Since the type terms required at run time are represented in **TYPEAREA**, we add two new tags `S_REF` and `S_BOTTOM` to the set of type tags, yielding

TTAGS = { `S_TOP, S_BOTTOM, S_MONO, S_REF, S_POLY` }

where `S_REF` corresponds to the subterm reference `STRUC` used in **DATAAREA** for ordinary terms. Together with the additional integrity constraints

if `tag(l) = S_REF` then `tref(l)` $\in$ **TYPEAREA**
`ttag(tref(l)) = S_POLY`
if `tag(l) = S_POLY` then `tref(l)` $\in$ **SORT**
`is_polymorphic(typeterm(l))`

the function

`typeterm:` **TYPEAREA** $\rightarrow$ **TYPETERM**

introduced in Section 6.3 is now completely defined by

$$\texttt{typeterm(l)} = \begin{cases} \texttt{TOP} & \text{if } \texttt{ttag(l)} = \texttt{S\_TOP} \\ \texttt{BOTTOM} & \text{if } \texttt{ttag(l)} = \texttt{S\_BOTTOM} \\ \texttt{tref(l)} & \text{if } \texttt{ttag(l)} = \texttt{S\_MONO} \\ \texttt{typeterm(tref(l))} & \text{if } \texttt{ttag(l)} = \texttt{S\_REF} \\ \texttt{s(a}_1\texttt{,...,a}_n\texttt{)} & \text{if } \texttt{ttag(l)} = \texttt{S\_POLY} \text{ and} \\ & \quad \texttt{s = tref(l)} \\ & \quad \texttt{n = sort\_arity(tref(l))} \\ & \quad \texttt{a}_i \texttt{ = typeterm(tref(l)+i)} \end{cases}$$

8.2 Creation of polymorphic type terms

We introduce a representation of polymorphic type terms occurring as arguments of the instructions in **CODEAREA** such that they can easily be loaded into **TYPEAREA**. For this purpose, we extend the compile function such that every polymorphic type term `tt` occurring in any of the generated PAM instructions introduced so far (i.e. `put_`, `get_`, `unify_variable`, respectively their refinements `put_free`, `put_mono` etc., see Section 7) is replaced by

`compile_type(tt)` $\in$ (**TTAG** $\times$ (**SORT** + **NAT**))*

Note that just for simplicity reasons this list representation abstracts away from the actual representation used in the PAM where the tagged type term representation occurring in the code is embedded into **CODEAREA**, mapping the list structure to the +-structure of **CODEAREA**. The function inverse to `compile_type` is defined by

$$\texttt{decompile\_type(L)} = \begin{cases} \texttt{TOP} & \text{if } \texttt{head(L)} = \texttt{<S\_TOP,.>} \\ \texttt{BOTTOM} & \text{if } \texttt{head(L)} = \texttt{<S\_BOTTOM,.>} \\ \texttt{s} & \text{if } \texttt{head(L)} = \texttt{<S\_MONO,s>} \\ \texttt{decompile\_type(}\underbrace{\texttt{tail(...(tail}}_{m-times}\texttt{(L))...))} & \text{if } \texttt{head(L)} = \texttt{<S\_REF,m>} \\ \texttt{s(a}_1\texttt{,...,a}_n\texttt{)} & \text{if } \texttt{head(L)} = \texttt{<S\_POLY,s>} \text{ and} \\ & \quad \texttt{n = sort\_arity(tref(l))} \\ & \quad \texttt{a}_i \texttt{ = decompile\_type(}\underbrace{\texttt{tail(...(tail}}_{i-times}\texttt{(L))...))} \end{cases}$$

and the integrity constraint we impose is

`decompile_type(compile_type(tt)) = tt`

for any type term `tt` $\in$ **TYPETERM**.

Using `compile_type(tt)` instead of `tt` itself passes this refined argument to the update `mk_unbound`. Since the update `mk_unbound` is defined in terms of `insert_type` which in turn is defined in terms of `insert_poly` for the polymorphic case, we only have to adapt the - until now - abstract update `insert_poly` (Section 6.4). It is now defined by

```
insert_poly(l,L) ≡ ref(l) := ttop
                   FORALL j = 1,...,length(L) DO
                      tval(ttop+j-1) := offset(ttop+j-1,nth(j,L))
                   ENDFORALL
                   ttop := ttop + length(L)
```

where

$$\texttt{offset(tl, <tag,k>)} = \begin{cases} \texttt{<tag, tl+k>} & \text{if } \texttt{tag} = \texttt{S\_REF} \\ \texttt{<tag, k>} & \text{otherwise} \end{cases}$$

POLYMORPHIC TYPE INSERTION LEMMA: The representation of type terms and the update defined above are a correct realization of the `insert_poly` update of Section 6.4, i.e. the POLYMORPHIC TYPE INSERTION CONDITION is satisfied.

Proof: The list representation generated by the function `compile_type` reflects exactly the structure of the representation of type terms in **TYPEAREA**, the only difference being that a sub-(type-)term pointer in **TYPEAREA** (with tag `S_REF`) is realized by an integer offset in the list representation. This representation difference is taken into account in the definition of `insert_poly` given above by adding the offset to the current **TYPEAREA** location in the `S_REF` case. ■

8.3 Polymorphic infimum

In order to refine the still abtract update $\texttt{poly\_infimum(l}_1\texttt{,l}_2\texttt{)}$ used in the Bind-2d rule of Section 6.5 to the infimum computation of polymorphic type terms as they occur in PROTOS-L, we need to know whether a type term is empty or not. For instance, given the standard notions of $\texttt{list(}\alpha_1\texttt{)}$ and $\texttt{pair(}\alpha_1\texttt{,}\alpha_2\texttt{)}$, `list(BOTTOM)` is not empty since it can be instantiated to the empty list `nil`, while `pair(BOTTOM,INTEGER)` is empty since there is no pair without a first component. The property that a type `tt` is not empty is formalized by the abbreviation

$$\texttt{inhabited(tt)} \quad \equiv \quad \texttt{solution(\{X:tt\})} \neq \texttt{nil}$$

where $\texttt{X} \in$ **VARIABLE**. Thus, from the conditions on the `solution` function in 6.1 we have e.g. `inhabited(BOTTOM)` = `false`, `inhabited(TOP)` = `true`, and furthermore `inhabited(list(BOTTOM))` = `true`, `inhabited(pair(BOTTOM,INTEGER))` = `false`.

We pose three additional integrity conditions. The first one requires that there are no 'empty' (monomorphic) sorts:

$$\texttt{is\_monomorphic(s)} \quad \Rightarrow \quad \texttt{inhabited(s)}$$

The second integrity constraint says that the infimum of polymorphic type terms is computed from the infimum of the argument types, and that it is always `BOTTOM` if we have different polymorphic types:

$\texttt{poly\_inf(s(tt}_1\texttt{,...,tt}_n\texttt{),s'(tt}_1\texttt{',...,tt}_n\texttt{')}$

$$= \begin{cases} \texttt{s(poly\_inf(tt}_1\texttt{,tt}_1\texttt{'),...,(poly\_inf(tt}_n\texttt{,tt}_n\texttt{'))} & \text{if } \texttt{s} = \texttt{s'} \\ & \text{and} \\ \quad \texttt{inhabited(s(poly\_inf(tt}_1\texttt{,tt}_1\texttt{'),...,poly\_inf(tt}_n\texttt{,tt}_n\texttt{')))} & \\ \texttt{BOTTOM} & \text{otherwise} \end{cases}$$

For the third integrity constraint we introduce a new abstract function

$$\texttt{inst\_modus:} \quad \textbf{SORT} \times \textbf{BOOL}^* \to \textbf{BOOL}$$

which tells whether terms of a given sort can be instantiated, depending only on the emptiness of the argument types, but not on the arguments themselves. This function specifies the 'instantiation modi' for a polymorphic sort, i.e. which type arguments of `s` may be `BOTTOM` so that `s` can still be instantiated. For instance, we have

inst_modus(list, [false]) = true
inst_modus(pair, [false, true]) = false

since

solution($\{$X:list(BOTTOM)$\}$) $\neq$ nil
solution($\{$X:pair(BOTTOM,INTEGER)$\}$) = nil

and thus

inhabited(list(BOTTOM)) = true
inhabited(pair(BOTTOM,INTEGER)) = false.

The general condition on inst_modus is

inst_modus(s,[b_1,...,b_n]) = true
$\Rightarrow$ (($\forall$ i $\in$ $\{$1,...,n$\}$. b_i = true $\Rightarrow$ inhabited(tt_i))
$\Rightarrow$ inhabited(s(tt_1,...,tt_n)))

For the realization of the poly_inf function in the PAM we introduce a new universe **P_NODE** that comes with a tree structure realized by the functions

p_root, p_current:	**P_NODE**
p_father:	**P_NODE** $\rightarrow$ **P_NODE**
p_sons:	**P_NODE** $\rightarrow$ **P_NODE***

where p_current is used to navigate through the tree. Each node in the **P_NODE** tree represents an infimum computation task for two type terms given as arguments, and it will be eventually be marked with the result. Thus, we have the three labelling functions

p_arg1, p_arg2:	**P_NODE** $\rightarrow$ **TYPEAREA**
p_result:	**P_NODE** $\rightarrow$ **TYPEAREA**

When a **P_NODE** element p represents the computation of the infimum of two polymorphic type terms typeterm(p_arg1(p)) = s(tt_1,...,tt_n) and typeterm(p_arg2(p)) = s(tt_1',...,tt_n'), then the n required computations of the infimum of the tt_i and tt_i' will correspond to the n nodes in the list p_sons(p). The **P_NODE** label

p_status: **P_NODE** $\rightarrow$ $\{$expand, expanded$\}$

indicates for each node whether the son nodes for it have still to be generated or not. The until now abtract update poly_infimum(l_1,l_2) for l_1, l_2 $\in$ **DATAAREA** is then defined by

poly_infimum(l_1,l_2) $\equiv$ p_arg1(p_root) := ref(l_1)
p_arg2(p_root) := ref(l_2)
p_status(p_root) := expand
p_current := p_root
p_return_arg := l_2
ll_what_to_do := polymorphic_infimum

It initializes the **P_NODE** tree containing just the root node. Additionally, it sets the new 0-ary function

p_return_arg : **DATAAREA**

which holds the location where the result of the polymorphic infimum computation will be written to when it has been finished.

ll_what_to_do $\in$ $\{$none, polymorphic_infiumum, polymorphic_propagation$\}$

is also a new 0-ary function that is added to the initial PAM algebras. Its initial value is none, indicating that no specific *l*ow-*l*evel actions have to be performed. All rules introduced so far get ll_what_to_do = none as an additional precondition; thus the definition of the poly_infimum(l_1,l_2) update just given blocks the applicability of all previous rules, until ll_what_to_do has been set back again to the value none by one of the rules to be introduced below. These new rules in turn will be guarded by the precondition

```
POLY-INF  ≡  OK & ll_what_to_do = polymorphic_infimum
```

(Note that such a scheme has been used before with the 0-ary function `what_to_do`, separating e.g. the binding and unification rules from all other rules.) Resetting of `ll_what_to_do` is done by means of the following abbreviation that holds for `tl` ∈ **TYPEAREA** and that is also used for the returning of values in intermediate stages of the polymorphic infimum computation:

```
p_return(tl) ≡ if p_current ≠ p_root
               then p_result(p_current) := tl
                    p_current := p_father(p_current)
               else ll_what_to_do := none
                    if ttag(tl) = S_BOTTOM
                    then backtrack
                    else bind_success
                         if ref(p_return_arg) ≠ tl
                           then trail(p_return_arg)
                                ref(p_return_arg) := tl
```

Note that the last if-then conditional is an optimization over the unconditional updates in the then-part since in case the return argument location `p_return_arg` already contains the required value we neither have to update nor to trail it.

Additionally, the following abbreviations will be used:

```
parg1 ≡ p_arg1(p_current)
parg2 ≡ p_arg2(p_current)
ttag1 ≡ ttag(parg1)
ttag2 ≡ ttag(parg2)
tref1 ≡ tref(parg1)
tref2 ≡ tref(parg2)
```

If either of the two type term arguments of `p_current` is `TOP` or `BOTTOM`, no son nodes have to be created and the result can be determined immediately since it is given by one of the two arguments.

Polymorphic Infimum 1 (S_TOP, S_BOTTOM)

```
if   POLY-INF
  & p_status(p_current) = expand
  & (ttag1 = S_TOP              | (ttag1 = S_BOTTOM
       OR                       |     OR
     ttag2 = S_BOTTOM)          |  ttag2 = S_TOP)
then
  p_status(p_current) := expanded
  p_return(parg2)               | p_return(parg1)
```

Also in the case of monomorphic types no son nodes have to be created.

Polymorphic Infimum 2 (S_MONO)

```
if   POLY-INF
  & p_status(p_current) = expand
  & ttag1 = S_MONO  &  ttag2 = S_MONO
  & subsort(tref1, | subsort(tref2, | sort_glb(tref1,tref2) | sort_glb(tref1,tref2)
            tref2) |          tref1) |    = BOTTOM           |    ≠ BOTTOM
then
  p_status(p_current) := expanded
  p_return(parg1)  | p_return(parg2)| make_s_bottom          | make_s_mono(
                   |                |                        | sort_glb(tref1,tref2))
                   |                | p_return(ttop)         | p_return(ttop)
```

where for s $\in$ **SORT** the allocation of new type locations in **TYPEAREA** is achieved by

```
make_s_mono(s) ≡ ttag(ttop) := S_MONO
                 tref(ttop) := s
                 ttop := ttop+
make_s_bottom  ≡ ttag(ttop) := S_BOTTOM
                 ttop := ttop+
```

If `p_current` points to a node with `S_POLY` tagged arguments for the first time (i.e. its status is `expand`), `sort_arity(tref(p_arg1(p_current)))` new son nodes are created and labelled accordingly (c.f. the integrity condition on `poly_inf` given above). `p_current` is set to the first of the new sons, and the new function

`p_rest_calls`: $\quad$ **P_NODE** $\rightarrow$ **P_NODE***

is set to the remaining son nodes, indicating that these nodes still have to be visited by `p_current`.

Polymorphic Infimum 3 (S_POLY-1)

```
if  POLY-INF
  & p_status(p_current) = expand
  & ttag1 = S_POLY  &  ttag2 = S_POLY
then
  p_status(p_current) := expanded
  LET n = sort_arity(tref1)
  extend P_NODE by temp(1),...,temp(n)
     where p_arg1(temp(i)) := parg1 + i
           p_arg2(temp(i)) := parg2 + i
           p_father(temp(i)) := p_current
           p_sons(p_current) := [temp(1),...,temp(n)]
           p_status(temp(i)) := expand
           p_current := temp(1)
           p_rest_calls(p_current) := [temp(2),...,temp(n)]
  endextend
```

When `p_current` points to a node with `S_POLY` tagged arguments for the second or a later time (i.e. its status is `expanded`) and there are still sons to be visited (i.e. `p_rest_calls(p_current))` $\neq$ `[]`), then `p_current` is set to the next son.

Polymorphic Infimum 4 (S_POLY-2)

```
if  POLY-INF
  & p_status(p_current) = expanded
  & ttag1 = S_POLY  &  ttag2 = S_POLY
  & p_rest_calls(p_current) ≠ []
then
  p_current := head(p_rest_calls(p_current))
  p_rest_calls(p_current) := tail(p_rest_calls(p_current))
```

When `p_current` points to a node with `S_POLY` tagged arguments for the second or a later time and all sons have already been visited (i.e. `p_rest_calls(p_current))` = `[]`), then all sub-computations for this node have been completed and the result is returned.

Polymorphic Infimum 5 (S_POLY-3)

```
if  POLY-INF
  & p_status(p_current) = expanded
  & ttag1 = S_POLY  &  ttag2 = S_POLY
```

```
  & p_rest_calls(p_current) = []
  & subtype(1)    | subtype(2)     | NOT (is_inhabited) | is_inhabited
then
  p_return(parg1) | p_return(parg2)| make_s_bottom      | write_poly_term
                  |                | p_return(ttop)     | p_return(ttop)
```

The three new abbreviations in the last rule are given by

```
subtype(i)      ≡ FOR ALL k = 1,...,sort_arity(tref1) .
                     pargi + k = p_result(nth(k,p_sons(p_current)))

write_poly_term ≡ tval(ttop) := tval(parg1)
                  FOR ALL k = 1,...,sort_arity(tref1) DO
                     tval(ttop + k) := tval(p_result(nth(k,p_sons(p_current))))
                  ENDFORALL
                  ttop := ttop + sort_arity(tref1) + 1

is_inhabited    ≡ inst_modus(tref1,[tb_1,...,tb_n])
```

where in the last abbreviation `n = sort_arity(tref1)`, and for `k = 1,...,n`

$$\mathtt{tb}_k \quad \equiv \quad \mathtt{ttag(p\_result(nth(k,p\_sons(p\_current))))} \neq \mathtt{S\_BOTTOM}$$

The subtype conditions in the above rule represent an optimization analogously to the subsort optimization in the `S_MONO` case (rule Polymorphic Infimum 2): only if the result differs from one of the two input arguments a *new* **TYPEAREA** location has to be returnd.

If `p_current` points to a node with `S_REF` tagged arguments for the first time (i.e. its status is `expand`), a single new son node labelled with the respective referenced type area locations is created.

Polymorphic Infimum 6 (S_REF-1)

```
if  POLY-INF
  & p_status(p_current) = expand
  & ttag1 = S_REF  &  ttag2 = S_REF
then
  p_status(p_current) := expanded
  extend P_NODE by temp
     where p_arg1(temp) := tref1
           p_arg2(temp) := tref2
           p_father(temp) := p_current
           p_sons(p_current) := [temp]
           p_status(temp) := expand
           p_current := temp
  endextend
```

When `p_current` points to a node with `S_REF` tagged arguments for the second time (i.e. its status is `expanded`), then the sub-computations for its single son node has been completed and the result is returned.

Polymorphic Infimum 7 (S_REF-2)

```
if  POLY-INF
  & p_status(p_current) = expanded
  & ttag1 = S_REF  &  ttag2 = S_REF
  & LET res = p_result(head(p_sons(p_current)))
  & res = tref1  | res = tref2    | ttag(res) = S_BOTTOM | ttag(res) ≠ S_BOTTOM
```

then

```
  p_return(parg1) | p_return(parg2) | p_return(res)          | make_s_ref(res)
                  |                 |                        | p_return(ttop)
```

where for tl $\in$ **TYPEAREA** the new abbreviation in the last rule is given by

```
make_s_ref(tl) ≡ ttag(ttop) := S_REF
                 tref(ttop) := tl
                 ttop := ttop+
```

POLYMORPHIC INFIMUM LEMMA: The polymorphic infimum rules given above are a correct realization of the poly_infimum(l_1,l_2) update of Section 6.5.

Proof: We have to show that the polymorphic infimum rules represent a correct realization of the poly_inf function on **TYPETERM** that is used in PROTOS-L (and which was introduced as an abtract function in Section 6.2). Taking the integrity constraints given for inf, sort_glb, and poly_inf in 6.1, 6.2, and 8.1 the proof follows by case analysis and induction on the sizes of typeterm(ref(l_1)) and typeterm(ref(l_2)). Note that the TRAILING CONDITION is also satisfied since in p_return(tl) the location p_return_arg (which had been set to l_2) is trailed if its value is to be changed. ■

8.4 Propagation of polymorphic type restrictions

The still abtract update poly_propagate(l_1,l_2) is used in the Bind-3b rule of Section 6.5 and in the Get-Structure-2b rule of Section 6.6. We refine this update to the propagation of polymorphic type constraints as they occur in PROTOS-L.

Let us start with an example. Consider the polymorphic declaration for list(α) with constructors

nil: $\rightarrow$ list(α)
cons: $\alpha \times$ list(α) $\rightarrow$ list(α)

and assume monomorphic types NAT and INTEGER with subsort(NAT,INTEGER) = true. Then solving the unification (or binding) constraint

X $\doteq$ cons(Y,L)

in the presence of the type prefix

{X:list(NAT), Y:INTEGER, L:list(INTEGER)}

generates the type constraint

cons(Y,L):list(NAT)

under the same type prefix. Thus, the update poly_propagate(l_1,l_2) would be called with term(l_2) = cons(Y,L) and typeterm(ref(l_1)) = list(NAT).

More generally, the arguments of the term referenced by l_2 (in the example Y:INTEGER and L:list(INTEGER)) must be restricted to the respective argument domains of the top-level functor f of term(l_2) (here: cons) where each type variable in an argument domain in the declaration of f (here: cons: $\alpha \times$ list(α) $\rightarrow$ list(α)) is replaced by the respective argument of typeterm(ref(l_1)) (here: replacing α by NAT, which yields cons: NAT $\times$ list(NAT) $\rightarrow$ list(NAT)).

This can be achieved in two steps: First, a new term f(X_1,...,X_m) (in the example: cons(X_1,X_2)) is created with appropriately type-restricted new variables X_i (here: X_1:NAT and X_2:list(NAT)), and second, this new term is unified with term(l_2). Thus, in the example the type constraint cons(Y,L):list(NAT) represented by poly_propagate(l_1,l_2) would be reduced to the unification problem

$$\texttt{cons(X}_1\texttt{,X}_2\texttt{)} \doteq \texttt{cons(Y,L)}$$

with type-constrained new variables $\texttt{X}_1$ and $\texttt{X}_2$. (In fact, this is a slight simplification of the representation over the actual PAM implementation where the top-level functor (here: cons) would not be generated since it is not needed; instead, the binding of the n argument variables of the new term can be called directly.)

For the general refinement of the polymorphic porpagation we assume as an integrity condition

$$\begin{aligned}\texttt{solution(}\{\texttt{f(t}_1\texttt{,...,t}_m\texttt{)}:\texttt{s(tt}_1\texttt{,...,tt}_n\texttt{)}\}\texttt{)} = \texttt{solution(}\{&\texttt{f(t}_1\texttt{,...,t}_m\texttt{)} \doteq \texttt{f(X}_1\texttt{,...,X}_m\texttt{)},\\ &\texttt{X}_1 : \texttt{subres(d}_1\texttt{,subst)}, \ \ldots,\\ &\texttt{X}_m : \texttt{subres(d}_m\texttt{,subst)}\}\texttt{)}\end{aligned}$$

where the $\texttt{X}_i$ are new variables, `f` has declaration

$$\texttt{f:} \quad \texttt{d}_1 \ \ldots \ \texttt{d}_m \ \rightarrow \ \texttt{s(}\alpha_1\texttt{,...,}\alpha_n\texttt{)} \in \texttt{prog}$$

and `subst` is the substitution (on type terms)

$$\texttt{subst} = \textstyle\bigcup_{k \in \{1,\ldots,n\}} \ \{\alpha_k \doteq \texttt{tt}_k\}$$

(c.f. [Bei90], [BMS91], [BM94]). Note that since $\texttt{s(tt}_1\texttt{,...,tt}_n\texttt{)}$ can not contain any type variables, also in $\texttt{subres(d}_j\texttt{,subst)}$ all type variables will have been replaced by ground type terms.

For the **SYMBOLTABLE** representation of the argument domains $\texttt{d}_j$ in a function declaration of the form given above we assume a compiled form similar to the representation of type terms in **CODEAREA** used in 8.2. We assume that the compiler numbers the variables in $\texttt{s(}\alpha_1\texttt{,...,}\alpha_n\texttt{)}$ from left to right, and use the additional tag `S_VAR` such that <S_VAR,k> represents the k-th variable α_k. Thus, the de-compilation of type terms in 8.2 is extended by

`decompile_type(L)` = α_k if head(L) = <S_VAR,k>

The function

$$\begin{aligned}\texttt{constr\_arg:} \quad & \textbf{SYMBOLTABLE} \times \textbf{NAT}\\ & \rightarrow ((\textbf{TTAG} + \{\texttt{S\_VAR}\}) \times (\textbf{SORT} + \textbf{NAT}))^*\end{aligned}$$

returns the argument domains $\texttt{d}_j$ for a constructor. For instance, given the above $\texttt{list(}\alpha\texttt{)}$ declaration, we have

```
constr_arg(entry(cons,2),1) = [<S_VAR,1>]
constr_arg(entry(cons,2),2) = [<S_POLY,list>, <S_VAR,1>]
```

More generally, for $\texttt{j} \in \{\texttt{1,...,m}\}$ we impose the integrity constraint

$$\texttt{decompile\_type(constr\_arg(entry(f,n),j))} = \texttt{d}_j$$

For the refinement of `poly_propagate` we add three new 0-ary functions to our initial PAM algebras: `pp_t` $\in$ **DATAAREA**, representing a reference to the term `t` to be retricted, `pp_tt` $\in$ **TYPEAREA**, a reference to the type term `tt` of the restriction, and `pp_i` $\in$ **NAT**, an index for the argument positions {1,...,m}. The update

```
poly_propagate(l1,l2)  ≡  pp_t := l2
                          pp_tt := ref(l1)
                          pp_i := 1
                          h ← <STRUC,h+>
                          val(h+) := ref(l2)
                          h := h++
                          ll_what_to_do := polymorphic_propagate
```

sets the three new 0-ary functions to their initial value, starts the generation of the new term by writing the top level functor on the heap, and blocks the applicability of all previous rules by updating `ll_what_to_do`. The following three polymorphic propagation rules are guarded by the condition `POLY-PROP` and use the abbreviations `hi` (for the heap location of the i-th argument of the term to be generated) and `pp_f` (for its top-level functor):

```
POLY-PROP ≡ OK & ll_what_to_do = polymorphic_propagate
hi        ≡ h + pp_i - 1
pp_f      ≡ ref(pp_t)
```

The first two propagation rules generate the argument variables $\mathtt{X}_1,\ldots,\mathtt{X}_m$. If there is still a variable to be generated (pp_i $\leq$ arity(pp_f)) and the (pp_i)th argument domain in the declaration of pp_f is not a type variable, then a variable with the respective type restriction is generated.

Polymorphic Propagation 1

```
if   POLY-PROP
   & pp_i ≤ arity(pp_f)
   & head(constr_arg(pp_f,pp_i)) =
        <S_TOP, .>  |   <S_MONO, s>     |   <S_POLY, .>
then
   tag(hi) := FREE  | tag(hi) := FREE_M | tag(hi) := FREE_P
                    | ref(hi) := s      | insert_poly(hi,
                    |                   |            constr_arg(pp_f,pp_i),
                    |                   |            pp_tt)
   pp_i := pp_i + 1
```

The update insert_poly(l,L,tl) is derived from its 2-argument counterpart in 8.2 by additionally substituting the (representation of the) type variable α_k by the (representation of the) k-th argument of typeterm(tl):

```
insert_poly(l,L,tl) ≡ ref(l) := ttop
                      FORALL j = 1,...,length(L) DO
                        tval(ttop+j-1) := offset&substitute(ttop+j-1, nth(j,L), tl)
                      ENDFORALL
                      ttop := ttop + length(L)
```

where

```
                                        ⎧ <tag, tl'+k>     if tag = S_REF
offset&substitute(tl', <tag,k>, tl) =  ⎨ tval(tl+k)       if tag = S_VAR
                                        ⎩ <tag, k>         otherwise
```

If there is still a variable to be generated (pp_i $\leq$ arity(pp_f)) and the (pp_i)th argument domain in the declaration of pp_f is a type variable (say, α_k), then the variable to be written on the heap must get the k-th type argument of typeterm(pp_tt) as its type restriction (i.e. tref(pp_tt + k)). If the latter is BOTTOM, backtrack update is executed since α_k:BOTTOM is an inconsistent type constraint (see 6.1).

Polymorphic Propagation 2

```
if   POLY-PROP
   & pp_i ≤ arity(pp_f)
   & head(constr_arg(pp_f,pp_i)) = <S_VAR, k>
   & ttag(pp_tt + k) =
       S_TOP        | S_MONO            | S_POLY            | S_BOTTOM
then
   tag(hi) := FREE | tag(hi) := FREE_M | tag(hi) := FREE_P | backtrack
                   |     ref(hi) := tref(pp_tt + k)        |
   pp_i := pp_i + 1                                        |
```

The third propagation rule is applied when all argument variables have been written on the heap (pp_i > arity(pp_f)). It is responsible for the unification of the term to be restricted (pp_t) with the newly generated term (referenced by h).

Polymorphic Propagation 3

```
if   POLY-PROP
  & pp_i > arity(pp_f)
then
  h := h + arity(pp_f)
  ll_what_to_do := none
  propagate_unify(h,pp_t)
```

with the abbreviations

propagate_unify(l_1,l_2)	≡ **if** still_unifying **then** push_on_unify_stack(l_1,l_2) **else** unify(l_1,l_2)
still_unifying	≡ what_to_do = Bind & return_from_bind = Unify
push_on_unify_stack(l_1,l_2)	≡ ref'(pdl++) := l_1 ref'(pdl+) := l_2 pdl := pdl++ what_to_do := Unify

Thus, if the machine is still in unifying mode, the update propagate_unify(l_1,l_2) just pushes the two locations to be unified onto the push down list **PDL** used for unification; otherwise the update unify(l_1,l_2) initializing unification is executed (see 3.2).

POLYMORPHIC PROPAGATION LEMMA: The polymorphic propagation rules given above are a correct realization of the poly_propagate(l_1,l_2) update of Section 6.5.

Proof: By induction on the number of arguments in typeterm(l_2) we can show that, from the time when ll_what_to_do is set to polymorphic_propagate to the time when the rule Polymorphic Propagation 3 is being executed, a term of the form f(X_1,...,X_m) is created on the heap. The rules Polymorphic Propagation 1 and 2 as well as the update insert_poly(l,L,tt) ensure that the proper type restrictions for X_i are inserted, i.e. - using the notation of the solution integrity constraint given in the beginning of this subsection - X_i : subres(d_i,subst). Note that if subres(d_i,subst) = BOTTOM, rule Polymorphic Propagation 2 carries out the backtrack update since solution({t : BOTTOM}) = nil for any term t.

Thus, we are left to show that also the equation part f(t_1,...,t_m) $\doteq$ f(X_1,...,X_m) is taken properly into account. This exactly is ensured by the updates of rule Polymorphic Propagation 3: By induction on the number of times the unification of the two terms to be unified will again cause a polymorphic propagation invocation, and using the UNIFICATION LEMMA of Section 3.2, we can show that at the time when the unification initiated by the update propagate_unify(h, pp_t) has been carried out (either with success or with failure) the post-conditions of the POLYMORPHIC PROPAGATION CONDITION are satisfied. ■

8.5 Main Theorem of Part II

Putting everything together, we obtain

Correctness Theorem 3 (Main Theorem of Part II): Compilation from PROTOS-L algebras to the PAM algebras with polymorphic, order-sorted type constraint handling is correct.

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.

[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.

[Bei90] C. Beierle. Types, modules and databases in the logic programming language PROTOS-L. In K. H. Bläsius, U. Hedtstück, and C.-R. Rollinger, editors, *Sorts and Types for Artificial Intelligence.* LNAI 418. Springer-Verlag, Berlin, 1990.

[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.

[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
  if solvable(cs ∪ {act ≐ rename(H,vi)})
  then cs  := cs ∪ {act ≐ rename(H,vi)}
       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
  if solvable(cs ∪ rename(P,vi))
  then cs  := cs ∪ rename(P,vi)
       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 of Part I

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:

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

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):

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

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

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

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 $\mathtt{g(s}_1\mathtt{,\ldots,s}_m\mathtt{)}$ is compiled:

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

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

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

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

with $\{Y_1,\ldots,Y_r\}$ 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
  & code(p) = put_value(l,x_j)
then
  x_j ← 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 $\mathtt{y}_1, \ldots, \mathtt{y}_n$ 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
y_i              ≡ e + 2 + i       ( 1 ≤ i ≤ stack_offset(cp) )
y_i(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 $\mathtt{x}_1, \ldots, \mathtt{x}_i$ of the current goal:

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

trust_me_else/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
```

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
```

switch_on_term

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