slightly modified version appeared in:

Formal Aspects of Computing, Vol. 8(5), pp. 539–564, 1996

Refinement of a typed WAM extension by polymorphic order-sorted types*

Christoph Beierle[1] and Egon Börger[2]

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

Keywords: Constraint logic programming, WAM, types, polymorphism, order-sorted types, correctness proof, evolving algebras

Abstract. We refine the mathematical specification of a WAM extension to type-constraint logic programming given in [BB96]. We provide a full specification and correctness proof of the PROTOS Abstract Machine (PAM), an extension of the WAM by polymorphic order-sorted unification as required by the logic programming language PROTOS-L, by refining the abstract type constraints used in [BB96] to the polymorphic order-sorted types of PROTOS-L. This allows us to develop a detailed and 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.

1. Introduction

In [BR95] a mathematical elaboration of Warren's Abstract Machine ([War83], [AK91]) for executing Prolog is given, coming in several refinement levels together with correctness proofs, and a correctness proof w.r.t. Börger's phenomenological Prolog description [Bör90]. In [BB96] we demonstrated how the evolving algebra approach naturally allows for modifications and extensions in the description of both the semantics of programming languages as well as of implementation

Address for correspondence: Christoph Beierle, Fachbereich Informatik, LG Praktische Informatik VIII, FernUniversität Hagen, Bahnhofstr. 48, D-58084 Hagen, Germany; e-mail: `christoph.beierle@fernuni-hagen.de`

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

methods. Extending Börger and Rosenzweig's WAM description, [BB96] provides a mathematical specification of a WAM extension to type-constraint logic programming and proves its correctness.

The reason that types are dealt with at the abstract machine level is that the extension of logic programming by types requires in general not only static type checking, but types may also be present at run time (see e.g. [MO84], [Han91], [Smo89]). In the presence of types and subtypes, restricting a variable to a subtype represents a constraint in the spirit of constraint logic programming. PROTOS-L [Bei92], 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 [BM94] that is an extension of the WAM by the required polymorphic order-sorted unification. Our aim is to provide here a full specification and correctness proof of the PAM, including extra-logical features and all WAM optimizations (like environment trimming and last call optimization), as well as PAM specific optimizations (like refined variable representation or switch on typed variables).

In [BB96] the notion of type constraints was deliberately kept abstract, in order to be applicable to a range of constraint formalisms such as Prolog III or CLP(R). Consequently, also on the abstract machine level, the type contraint solving parts had to be kept abstract. In this paper we refine these abstract type constraints to the polymorphic order-sorted types of PROTOS-L. We do this again in several refinement steps. This allows us to develop a detailed and mathematically precise account of the PAM's compiled type constraint representation and solving facilities, and to prove its correctness w.r.t. PROTOS-L.

Section 2 introduces the representation and constraint solving of monomorphic, order-sorted type constraints. Section 3 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 4 gives a detailed account of polymorphic type constraint representation and solving in the PAM. Since this paper is a direct sequel to [BB96], we assume the reader to be familiar with it and refer to it for unexplained definitions and notations and for further references to the literature.

2. PAM algebras with monomorphic type constraints

2.1. Binding

We start with 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 $\mathtt{l}_1,\mathtt{l}_2 \in$ **DATAAREA** we define

```
bind($\mathtt{l}_1$,$\mathtt{l}_2$)  $\equiv$ arg1 := $\mathtt{l}_1$
```

```
                  arg2 := $l_2$
                  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($l_1$,$l_2$)   ≡ ref¨(tr) := ($l_1$, val($l_1$))
                  ref¨(tr+):= ($l_2$, val($l_2$))
                  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(l_1) there are two alternative conditions on the update occur_check(l_1,l_2), 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(l_1,l_2) is empty; otherwise it has the following effect: If mk_var(l_1) is among the variables of term(l_2) 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

inf: **TYPETERM** × **TYPETERM** → **TYPETERM**

which yields the *infimum* of two type terms, which may also be BOTTOM ∈ **TYPETERM**. TOP and BOTTOM can be thought of as ‘maximal’ and ‘minimal’ type terms. As integrity constraints we have

```
inf(TOP,tt) = inf(tt,TOP) = tt
inf(BOTTOM,tt) = inf(tt,BOTTOM) = BOTTOM
solution({t:BOTTOM}) = nil
solution({X:$tt_1$, X:$tt_2$}) = solution({X:inf($tt_1$,$tt_2$)})
```

for any t ∈ **TERM** and tt_i ∈ **TYPETERM**.

Bind-2 (Bind-Var-Var)

```
if  BIND
  & unbound(arg2)
  & LET inf = inf(ref(arg1),ref(arg2))
  & inf $\neq$ BOTTOM                           | inf = BOTTOM
  & inf $\neq$ ref(arg2)     | inf = ref(arg2) |
then
  trail(arg1,arg2)        | trail(arg1)     | backtrack
  insert_type(arg2,inf)   |                 |
  arg1 $\leftarrow$ <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 get_structure(f,x_i) when the dereferenced value of 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 l_1, l_2, $l \in$ **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 propagate(l_1,l_2) we have if unbound(l_1), ref(l_1) $\in$ **TYPETERM**, tag(l_2) = STRUC, and term(l_2) $\in$ **TERM**:

```
LET CS = {term($l_2$):ref($l_1$)}
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 $\leftarrow$ <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 in [BB96], 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 bind(l_1,l_2) is by case analysis and induction on the size of term(l_2), 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). □

2.2. Monomorphic, order-sorted types

Before introducing a representation for type terms we introduce some new functions and universes that are related to **TYPETERM**. Until now we have kept **TYPETERM** abstract; in this section we come to some more specific type term characteristics such as monomorphic and polymorphic type terms. Going by stepwise refinement, 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 **TYPETERM** we introduce the three functions

$$\texttt{is\_top, is\_monomorphic, is\_polymorphic:}\ \mathbf{TYPETERM} \rightarrow \mathbf{BOOL}$$

with their obvious meaning. The function

$$\texttt{target\_sort:}\ \mathbf{SYMBOLTABLE} \rightarrow \mathbf{SORT}$$

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

$$\texttt{subsort:}\ \mathbf{SORT} \times \mathbf{SORT} \rightarrow \mathbf{BOOL}$$

defining the order relation on the monomorphic sorts (and being undefined on the polymorhic sorts [Bei92]). For the refinement of type constraint handling we assume two functions

$$\begin{array}{l}\texttt{sort\_glb:}\ \mathbf{SORT} \times \mathbf{SORT} \rightarrow \mathbf{SORT}\\ \texttt{poly\_inf:}\ \mathbf{TYPETERM} \times \mathbf{TYPETERM} \rightarrow \mathbf{TYPETERM}\end{array}$$

that refine the `inf` function (from 2.1) in the sense that for any $\texttt{tt}_1$, $\texttt{tt}_2 \in$ **TYPETERM**

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

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

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

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.

2.3. 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 4):

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

ttop, tbottom, TOP: → TYPEAREA
+, -: TYPEAREA → TYPEAREA
tval: TYPEAREA → TO
```

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

```
ttag: TO → TTAGS
tref: TO → SORT + TYPEAREA
```

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

```
{ S_TOP, S_MONO, S_POLY } ⊆ 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) ∈ SORT
                            is_monomorphic(tref(l))
if ttag(l) = S_POLY   then  is_polymorphic(tref(l))
```

where the auxiliary function `typeterm`: **TYPEAREA** → **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 in [BB96] by replacing the universe **TYPETERM** by its representing universe **TYPEAREA**. The codomain of the `ref` function (from 3.1 in [BB96]) now contains **TYPEAREA**, and in the integrity constraints of 3.1 in [BB96] 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` ∈ **DATAAREA** we therefore use the following abbreviations:

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

2.4. 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 in [BB96]) (Its use in the multiple `mk_unbounds` update in `get_structure` will be refined in Section 2.6 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 ∈ **DATAAREA**.[1]

Similarly, the insertion of polymorphic type terms by insert_poly(l,tt) will be handled in Section 4. 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 ∈ **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 ∈ **TYPETERM** with is_polymorphic(tt):

$$\texttt{(term', prefix')} = \texttt{conres(term,prefix}\backslash\texttt{mk\_var(l}_1\texttt{),}\{\texttt{mk\_var(l}_1\texttt{):tt}\}\texttt{)}$$

TYPE INSERTION LEMMA: The refinement of the insert_type update satisfies the TYPE INSERTING CONDITION of 3.5 in [BB96].

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

2.5. Binding of type constrained variables

We refine the binding rules of Section 2.1 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))
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
```

[1] 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.

```
  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)
```

For the still abstract update poly_infimum(l_1,l_2) used when binding two polymorphically restricted variables 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 ≠ val' then the location l will be trailed
    else  backtrack update will be executed
```

Rule Bind-3 of Section 2.1 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) |
if   BIND                                          Bind-3b (Bind-Poly-Struc)
  & 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 2.1 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 correctly realize the binding rules of Section 2.1 and thus the binding update of 3.2 in [BB96].

Proof. Following the proof of the BINDING LEMMA in 2.1 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 in [BB96]). For the monomorphic case we conclude from the last integrity constraint given in Section 2.2 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. □

2.6. Getting of structures

We refine the get_struture rules of Section 3.4 in [BB96] according to the type term representation. Rule Get-Structure-1 remains unchanged. Get-Structure-2 for the case that x_i is an unbound variable is replaced by the following rules:

```
if   RUN                                                  Get-Structure-2a
   & code(p) = get_structure(f,x_i)
   & monomorphic(deref(x_i))
   & NOT ( subsort(target_sort(f),sort(deref(x_i))) )
then
   backtrack
```

```
if   RUN                                                  Get-Structure-2b
   & code(p) = get_structure(f,x_i)
   & top(deref(x_i))            | polymorphic(deref(x_i))
         OR                     |
     (monomorphic(deref(x_i)) & |
      subsort(target_sort(f),   |
              sort(deref(x_i))) |
then
   h ← <STRUC,h+>
   bind(deref($x_i$),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($x_i$))
   succeed
```

Thus, the only remaining abstract update is in the case when 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 in [BB96], 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 in [BB96] 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 in [BB96] are propagated onto the arguments of the structure. □

Whereas we have now 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(l_1,l_2), and poly_propagate(l_1,l_2) which will be refined in Section 4.

3. PAM Optimizations

3.1. 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 [BM94]. 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 in [BB96]) to include also **SORT**. Let l ∈ **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 in [BB96] 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

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

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 2.6).

Similarly, the rules for initializing variables (Section 5.2 in [BB96]) 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 $\texttt{get\_variable(l,x}_j\texttt{,tt)}$ is replaced by one of the three new instructions

$\texttt{get\_free(l,x}_j\texttt{)}$ $\texttt{get\_mono(l,x}_j\texttt{,tt)}$ $\texttt{get\_poly(l,x}_j\texttt{,tt)}$

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.

```
if   RUN                                                 Put-1 (X variable)
  & 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)
  $x_i$ $\leftarrow$ <REF,h>
  $x_j$ $\leftarrow$ <REF,h>
  h := h+
  succeed
```

```
if   RUN                                                 Put-2 (Y variable)
  &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)
  $x_j$ $\leftarrow$ <REF,$y_n$>
  succeed
```

```
if   RUN                                                 Get (Variable)
  & code(p) =
      get_free(l,$x_j$) | get_mono(l,$x_j$,s)     | get_poly(l,$x_j$,tt)
then
  l $\leftarrow$ $x_j$              | mk_unbound_mono(l,s)| mk_unbound_poly(l,tt)
                      | bind(l,$x_j$)          | bind(l,$x_j$)
  succeed
```

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

```
if   RUN                                                 Unify (Write Mode)
  & 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 $\leftarrow$ <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 2.

Proof. The only type inserting update of 2.4 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 2.4. Trailing still works fine since in 4.2 in [BB96] we trailed the complete `val` decoration of a data area location - including its tag - and restored it upon backtracking. With these two observations the proof follows by case analysis for the three different kinds of type restrictions. Showing that each variable is initialized properly is straightforward; the correct treatment of the 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)`. □

3.2. 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 in [BB96]) which increases the determinacy detection abilities. Since only monomorphic types can have explicitly defined subtypes there are two switch-on-term instructions. (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, leading to additional parameters in the following instructions.)

```
if   RUN                                                    Switch-on-poly-term
  & code(p) = switch_on_poly_term(i,Lfree,Lstruc)
  & tag(deref($x_i$)) $\in$ {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. Appendix B.7 in [BB96]) 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 [Bei92].

```
if   RUN                                                    Switch-on-mono-term
  & 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** × **CODEAREA**, and select$_{sort}$(Table,s) yields the location c such that (s,c) is in Table.

For the correctness proof for the extended switching instructions we must extend the assumptions on the compiler stated in 2.2 in [BB96]. The defintion of chain is changed so that the two cases for switch_on_term are replaced by

```
chain(Ptr) =
  { chain(Lf)    if code(Ptr) = switch_on_poly_term(i,Lf,Ls)
                    and is_top(X$_i$) or is_polymorphic(X$_i$)
    chain(Ls)    if code(Ptr) = switch_on_poly_term(i,Lf,Ls)
                    and is_struct(X$_i$)
    chain(Lf)    if code(Ptr) = switch_on_mono_term(i,Lf,Lfm,Ls)
                    and is_top(X$_i$)
    chain(Lfm)   if code(Ptr) = switch_on_mono_term(i,Lf,Lfm,Ls)
                    and is_monomorphic(X$_i$)
    chain(Ls)    if code(Ptr) = switch_on_mono_term(i,Lf,Lfm,Ls)
                    and is_struct(X$_i$)
    chain(select$_{sort}$(T,s))   if code(Ptr) = switch_on_sort(i,T)
                                   and s = sort(X$_i$)
    ...
```

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. □

The special representation of typed variables introduced in this section yield that the type extension in the PAM is 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 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.

4. 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.

4.1. Representation of polymorphic type terms

For the representation of polymorphic type terms we introduce the function

sort_arity: **SORT** → **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 in [BB96]) and the functions on **SORT** is regulated by the following integrity constraints: For each function declaration of the form

f: $d_1 \ldots d_m \rightarrow s(\alpha_1,\ldots,\alpha_n)$

with m, n ≥ 0, pairwise distinct (type) variables α_i that occur in $d_1,\ldots,d_m$, and each tt = s(...) ∈ **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
```

We illustrate these integrity constraints by an example. 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    arity(entry(succ,1))   =1
target_sort(entry(cons,2))   =list   arity(entry(cons,2))   =2
target_sort(entry(mk_pair,2))=pair   arity(entry(mk_pair,2))=2

sort_arity(nat)  = 0      is_monomorphic(nat)          = true
sort_arity(list) = 1      is_polymorphic(list(list(γ))) = true
sort_arity(pair) = 2
```

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) ∈ TYPEAREA
                           ttag(tref(l)) = S_POLY
if tag(l) = S_POLY   then  tref(l) ∈ SORT
                           is_polymorphic(typeterm(l))
```

the function

typeterm: **TYPEAREA** → **TYPETERM**

introduced in Section 2.3 is now completely defined by

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

4.2. 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 3) is replaced by

$$\texttt{compile\_type(tt)} \in \texttt{(}\mathbf{TTAG} \times \texttt{(}\mathbf{SORT} \texttt{ + } \mathbf{NAT}\texttt{))}^*$$

For simplicity this list representation abstracts 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

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

For any type term $\texttt{tt} \in \mathbf{TYPETERM}$ we impose the integrity constraint

`decompile_type(compile_type(tt)) = tt`

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 2.4). 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 2.4, 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. □

4.3. 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 2.5 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

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

where X ∈ **VARIABLE**. Thus, from the conditions on the solution function in 2.1 we have e.g. inhabited(BOTTOM) = false, inhabited(TOP) = true, 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

$$\texttt{inst\_modus:} \ \mathbf{SORT} \times \mathbf{BOOL}^* \rightarrow \mathbf{BOOL}$$

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_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  $\equiv$  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) $\equiv$ if p_current $\neq$ 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) $\neq$ 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 for $i = 1, 2$:

```
parg$i$ $\equiv$ p_arg$i$(p_current)
ttag$i$ $\equiv$ ttag(parg$i$)
tref$i$ $\equiv$ tref(parg$i$)
```

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.

```
if   POLY-INF                         Polymorphic Infimum 1 (S_TOP, S_BOTTOM)
  & 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.

```
if   POLY-INF                                Polymorphic Infimum 2 (S_MONO)
  & p_status(p_current) = expand
  & ttag1 = S_MONO  &  ttag2 = S_MONO
  & subsort(tref1,| subsort(tref2, | sort_glb(tref1,| sort_glb(tref1,
            tref2)|         tref1) |          tref2)|          tref2)
```

```
                    |                | = 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: **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) = | ttag(res) ≠
                 |                |    S_BOTTOM |     S_BOTTOM
then
  p_return(parg1) | p_return(parg2) | p_return(res)| make_s_ref(res)
                  |                 |              | p_return(ttop)
```

where for tl ∈ **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 2.5.

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 2.2). Taking the integrity constraints given for inf, sort_glb, and poly_inf in 2.1, 2.2, and 4.1 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. □

4.4. Propagation of polymorphic type restrictions

The still abtract update poly_propagate(l_1,l_2) is used in the Bind-3b rule of Section 2.5 and in the Get-Structure-2b rule of Section 2.6. 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: → list(α)
cons: α × list(α) → 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: α × list(α) → list(α)) is replaced by the respective argument of

$\texttt{typeterm(ref(l}_1\texttt{))}$ (here: replacing α by NAT, which yields $\texttt{cons: NAT} \times \texttt{list(NAT)} \rightarrow \texttt{list(NAT)}$).

This can be achieved in two steps: First, a new term $\texttt{f(X}_1\texttt{,...,X}_m\texttt{)}$ (in the example: $\texttt{cons(X}_1\texttt{,X}_2\texttt{)}$) is created with appropriately type-restricted new variables $\texttt{X}_i$ (here: $\texttt{X}_1 \texttt{:NAT}$ and $\texttt{X}_2\texttt{:list(NAT)}$), and second, this new term is unified with $\texttt{term(l}_2\texttt{)}$. Thus, in the example the type constraint $\texttt{cons(Y,L):list(NAT)}$ represented by $\texttt{poly\_propagate(l}_1\texttt{,l}_2\texttt{)}$ 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

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

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

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

and subst is the substitution (on type terms)

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

(c.f. [Bei92], [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 4.2. 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 4.2 is extended by

$$\texttt{decompile\_type(L) = } \alpha_k \qquad \text{if } \texttt{head(L) = <S\_VAR,k>}$$

The function

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

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

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

For the refinement of poly_propagate we add three new 0-ary functions to our initial PAM algebras: pp_t ∈ **DATAAREA**, representing a reference to the term t to be retricted, pp_tt ∈ **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(l_1,l_2) ≡ pp_t := l_2
                          pp_tt := ref(l_1)
                          pp_i := 1
                          h ← <STRUC,h+>
                          val(h+) := ref(l_2)
                          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 $X_1,\ldots,X_m$. If there is still a variable to be generated (pp_i ≤ 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 4.2 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&subst(ttop+j-1, nth(j,L), tl)
      ENDFORALL
      ttop := ttop + length(L)
```

where

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

If there is still a variable to be generated (pp_i ≤ 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 2.1).

```
if   POLY-PROP                                          Polymorphic Propagation 2
  & 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).

```
if   POLY-PROP                                          Polymorphic Propagation 3
  & 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(l1,l2)       ≡ if still_unifying
                                     then push_on_unify_stack(l1,l2)
                                     else unify(l1,l2)

    still_unifying               ≡ what_to_do = Bind &
                                   return_from_bind = Unify

    push_on_unify_stack(l1,l2)   ≡ ref'(pdl++) := l1
                                   ref'(pdl+) := l2
                                   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 in [BB96]).

POLYMORPHIC PROPAGATION LEMMA: The polymorphic propagation rules given above are a correct realization of the poly_propagate(l_1,l_2) update of Section 2.5.

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 $\mathtt{X}_i$ are inserted, i.e. - using the notation of the `solution` integrity constraint given in the beginning of this subsection - $\mathtt{X}_i : \mathtt{subres(d}_i\mathtt{,subst)}$. Note that if $\mathtt{subres(d}_i\mathtt{,subst) = BOTTOM}$, 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 $\mathtt{f(t}_1\mathtt{,\ldots,t}_m\mathtt{)} \doteq \mathtt{f(X}_1\mathtt{,\ldots,X}_m\mathtt{)}$ 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 in [BB96], 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. □

4.5. Main Theorem

Putting everything together, we obtain

Correctness Theorem 3: 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.

[BB96] C. Beierle and E. Börger. Specification and correctness proof of a WAM extension with abstract type constraints. *Formal Aspects of Computing*, 8(4), 1996.

[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.

[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.

[Bör90] E. Börger. A logical operational semantics of full Prolog. Part I. Selection core and control. *CSL'89 - 3rd Workshop on Computer Science Logic*. LNCS 440, pages 36–64. Springer-Verlag, Berlin, 1990.

[BR95] E. Börger and D. Rosenzweig. The WAM – definition and compiler correctness. In C. Beierle and L. Plümer, editors, *Logic Programming: Formal Methods and Practical Applications*, Studies in Computer Science and Artificial Intelligence, chapter 2, pages 20–90. Elsevier Science B.V./North-Holland, Amsterdam, 1995.

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

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

[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.