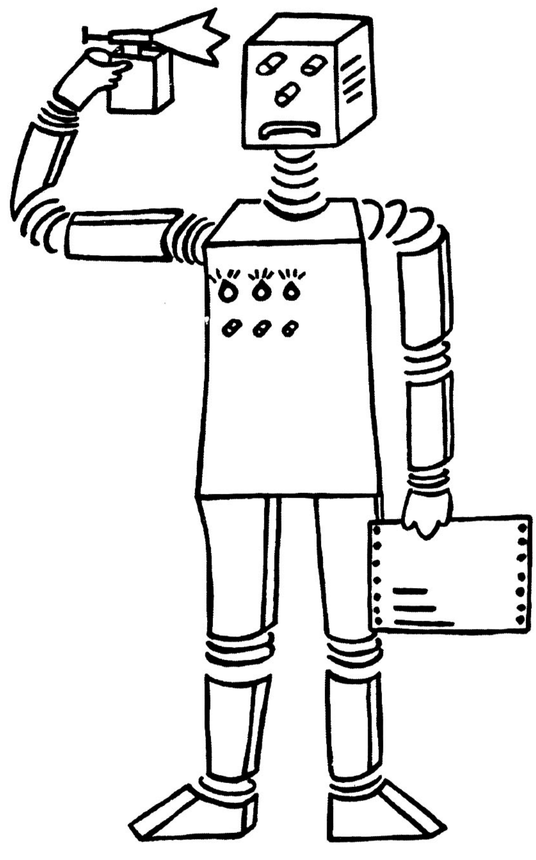
SEKI-Working Paper

Fachbereich Informatik
Universität Kaiserslautern
Postfach 3049
D-6750 Kaiserslautern 1, W. Germany



Extending the WARREN Abstract Machine to Feature Prolog

Peter Forster

SEKI-Working Paper SWP-87-10 Dez. 87

Extending the WARREN Abstract Machine to Feature Prolog

Peter Forster

SEKI-Working Paper SWP-87-10 Dez. 87

Extending the WARREN Abstract Machine to Feature Prolog

Peter Forster

Fachbereich Informatik
Universität Kaiserslautern
Postfach 3049
D-6750 Kaiserslautern
W.Germany

Abstract:
Inheritance hierarchies are employed in knowledge representation and object-oriented programming in the sense of representing taxonomic information. Feature Prolog provides a useful tool to represent taxonomic information in logic in a simple and natural way. In our approach, inheritance hierarchies are built-up from feature types, that are record-like structures, ordered by subtyping. The presence of feature types reduces the deduction tree and avoids unnecessary backtracking. In Feature Prolog there are feature terms besides the common Prolog terms - used to denote subsets of feature types. The integration of feature terms into the Prolog inference mechanism needs an extension of SLD-resolution with feature unification, that is unification respecting the taxonomic information of the feature types. We describe an extension of the abstract Prolog instruction set, known as WARREN Abstract Machine, for inheritance hierarchies.

1.Introduction and Motivation

What is the intention to deal with inheritance hierarchies in Prolog or more generally in deduction systems? Ait-Kaci /Ait-Kaci 85/ writes: "Since the early days of research in Automated Deduction, inheritance has been proposed as a means to capture a special kind of information, viz., taxonomic information. For example, when it is asserted that *whales are mammals*, we understand that whatever properties mammals possess should also hold for whales."

Naturally, this meaning of inheritance can be expressed in predicate logic by the implication.

$$\forall x.\ Whale(x) \rightarrow Mammal(x)$$

It is semantically correct to solve this in first order logic by a deduction step, but is this pragmatically useful? Isn't it possible to find the information that *whales* are *mammals* without a deduction step to reduce the search space? Is it possible to integrate this kind of taxonomic information directly in deduction systems, namely Prolog? Feature Prolog is an approach to achieve that.

In the following we give an extension of Prolog with inheritance hierarchies, as a system of record-like structures, called feature types, ordered by subtyping /Smolka Ait-Kaci 87/. Feature types are similiar to frames, in the sense that every feature type has a set of features (slots); and each feature is an access function to one slot of the feature type. Sometimes we use the notion *class* in this paper as a synonym for feature type - being more meaningful from the history in connection with object orientated programming. An *instance* is an object, that belongs to a class. In Feature Prolog we also call an instance feature constant.

Now we can describe ***Whale*** as a subclass of the class ***Mammal***. ***Whale*** inherites all features of the class ***Mammal***. For example the inherite feature ***biotope*** is constrained to ***ocean***. This kind of taxonomic information could be represented in first order logic by deduction, but in the following examples we will see that this is a very inefficient method. A better way is, to built this taxonomic information directly into deduction systems.

Many proposals have been offered to deal with taxonomic information. /DeKo 79/ und /AlFr 82/ transform the taxonomic information directly into first order logic, but then we have the disadvantages pointed out above. /BriFi 83/ und /SkiMi 79/ interprete taxonomic information as semantic nets, but the corresponding implementation doesn't have the power of a Prolog like description language. Feature Prolog is similar to Ait-Kaci and Nasr's Prolog dialect LOGIN, but there are two differences. First, LOGIN has only feature types (called Ψ-types in LOGIN), while Feature Prolog accommodates both feature and constructor types. Consequently, Feature Prolog's unification, called feature unification /Smolka Ait-Kaci 87/, combines LOGIN's Ψ-unification with order-sorted unification /Sch 85/, /Wal 87/. Furthermore, while Ψ-unification admits cyclic structures, feature unification disallows them. Second, LOGIN does not force the programmer to declare which features are possible for a feature type and thus has a weaker type checking discipline.

The computational power of Prolog and Feature Prolog is the same, but with the integration of taxonomic information and a modified unification algorithm - using the taxonomic information effectively, unnecessary backtracking is avoided and therefore the search space will be reduced.

We give an example to show how laboriously and inefficient taxonomic information is represented in common Prolog.

We want to know if the sign_of_zodiac of Peter's grandfather is pisces. We have some rules for *grandfather* and *sign_of_zodiac*. Knowledge about Peter, Mary, ... to be persons and january, february, ... to be months is stored explicitely as facts.

The relations *has_father* and *month_of_birth* are also defined as facts.

Example 1a

Z is grandfather of Y, if X,Y, and Z are persons and the father of Y is Z. Y has to be the father of X or Y is the mother of X. The Prolog system achieves all informations by searching in the database. E.g. the information that the father of Peter is Bill gets the Prolog system after searching in the database of *has_father*(...,...).

```
% THE RULES FOR grandfather %
     grandfather(X,Z):- person(X),person(Y),person(Z),
          has_father(X,Y),has_father(Y,Z).
     grandfather(X,Z):- person(X),person(Y),person(Z),
          has_mother(X,Y),has_father(Y,Z).
```

Y is sign of zodiac of X, if X is a person and X is born in a special month Z. The whole information about X achieves the Prolog system by *searching* in the database of *person*(...) and *month_of_birth*(...,...).

```
% THE RULES FOR sign_of_zodiac %
     sign-of-zodiac(X,capricorn):-
          person(X),month_of_birth(X,january).
     sign-of-zodiac(X,aquarius):-
          person(X),month_of_birth(X,february).
     sign-of-zodiac(X,pisces):-
          person(X),month_of_birth(X,march).
     .....

% THE DATABASE %
     person(anne).
     person(mary).
     person(peter).
     person(bill).
     person(john).
     .....
     month_of_birth(peter,august).
```

```
month_of_birth(john,march).
month_of_birth(bill,january).
month_of_birth(mary,february).
month_of_birth(cristine,december).
......
has_father(christine,john).
has_father(mary,bill).
has_father(bill,john).
has_father(peter,bill).
.....
has_mother(john,christine).
.....
```

```
% THE QUERY: %
    ?- grandfather(peter,X),sign_of_zodiac(X,pisces).
```

Have a lot of fun and time to get the correct solution for this query.

The two main disadvantages of this Prolog program are:

- A lot of unnecessary backtracking has to be executed in order to instantiate the variable X of the query with the grandfather of *peter*, whose sign of zodiac is *pisces*.
- A naive reader may have difficulties to understand the semantic of the clauses defining *grandfather(X,Y)*.

Example 1b presents a more natural representation of the above information in a Prolog like syntax, with the meaning that Peter, Mary, ... are individuals (constants) of type *person*. January, february, march, ... are constants of type *month*.

Example 1b

```
% THE DECLARATION PART: %
    csort(peter,person).
    csort(bill,person).
    csort(john,person).
    .....
    csort(january,month).
    csort(february,month).
    csort(march,month).
    .....
```

Each person has a person as mother and another person as father. Each person is born in a special month.

```
% THE FEATURES OF THE FEATURE TYPE person %
    has-feature(person,mother,person).
    has-feature(person,father,person).
    has-feature(person,month_of_birth,month).
```

The father of Peter is Bill and Bill's father is John. Bill is born in january and John in march.

```
% THE FEATURES OF THE FEATURE CONSTANTS %
    has-feature(peter,father,bill).
    has-feature(bill,father,john).
    has-feature(bill,month_of_birth,january).
    has-feature(john,month_of_birth,march).
    .....
```

Z is the grandfather of a person X, whose father or mother is a person Y, whose father is the person Z. The Feature Prolog system achieves all information by the features *father* and *mother*, that compute the value of someones father or mother. Grandfather(X,Z) gets the information that Z is the grandfather of X by this computation instead of searching.

```
% THE RULE BASE %
% THE RULES FOR grandfather %
    grandfather(X:person{father ->Y:person{father ->Z:person}},
                Z).
    grandfather(X:person{mother ->Y:person{father ->Z:person}},
                Z).
```

The information about someones sign of zodiac is achieved by the feature *month_of_birth*.

```
% THE RULES FOR sign of zodiac %
    sign-of-zodiac(X:person{month_of_birth->january},capricorn}.
    sign-of-zodiac(X:person{month_of_birth->february},aquarius}.
    .....

% THE QUERY: %
    ?-grandfather(peter,X:person),sign_of_zodiac(X,pisces).
```

The relation that a feature type is a subtype of another feature type is expressed with the built-in predicate *subsort*. The semantic of *csort(const,type)* is that the constant *const* is an individual of the feature type *type*. Notice, that constants can also be considered as feature types, consisting of exactly this constant only. *has-feature(tn,F,tm)* assigns the type *tn* a feature *F* with type *tm*. Note that all subtypes of *tn* and hence all individuals of type *tn*, inherite the feature *F* with type *tm* or if the feature *F* is defined in a subtype of *tn* then the type of *F* has to be a subtype of *tm*.
The relation that Peter's father is bill, is implicitly given by *has-feature(peter,father,bill)*. Also the knowledge that Peter, Mary, ... are persons and that january,february, ... are month. It is not necessary to define it in the database. If the goal *grandfather* with the constant *peter* as first argument and the variable *X* of type *person* as second argument is executed, the feature *father* selects *Bill* as Peter's father. This kind of dealing with implicite knowledge <u>replaces searching by effective computation</u>, hence we reduce the deduction tree and avoid unnecessary backtracking. The following figures 1 and 2 shows the type information given in the above program and the solution of the query one more graphically.

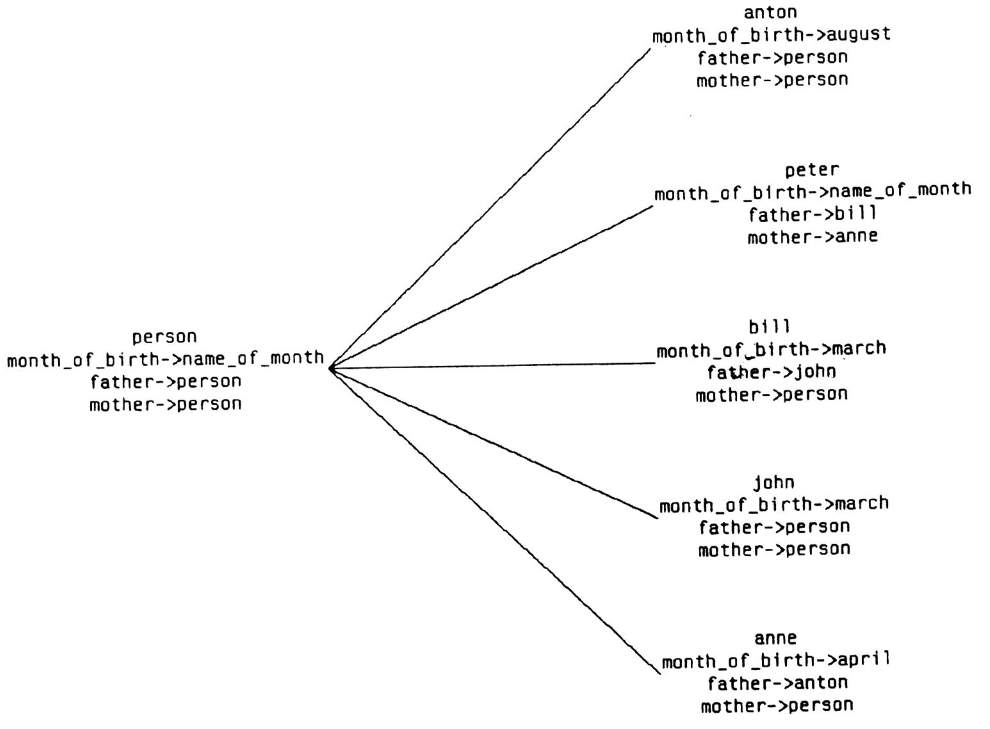


Fig.1

The ordering of the feature types and the set of features for the feature type *person*.

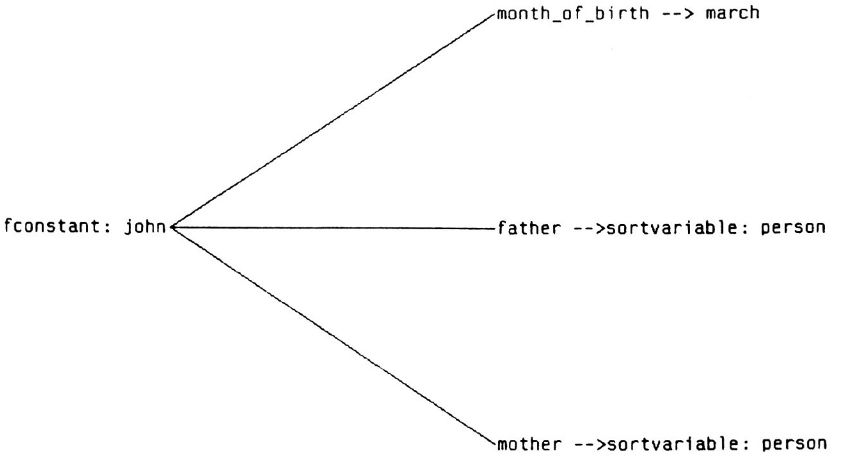


Fig.2

The solution of the query. The variable X is instantiated with John, because John is the grandfather of Peter and John is born in the month march so that his sign of zodiac is pisces.

2. *Feature Unification in Prolog*

We give a short outline of the computational part of inheritance hierarchies called feature unification, that extends common Prolog unification for feature terms. A formal definition and investigation of feature unification is given in /Smolka Ait-Kaci 87/.

A term is a variable, a constant or a structure. If a variable has no reference to a term we call this variable an *unbound* variable.

In common Prolog two terms are unifyable if one of the following conditions succeeds:

(1) If one term is an unbound variable, then the other term is bound to this unbound variable.
(2) If both terms are constants, then they have to be identical.
(3) If both terms are structures, the two functors have to be identical and the arguments of the structures have to be unifyable.

For a more formal approach we assume 2 pairwise disjoint, countably infinite sets of symbols:

- Functions Symbols: (f,g,h)

  Every function symbol f has an arity, which is an nonnegative integer; function symbols with arity zero are called constant symbols.

- Variables: (X,Y,Z).

A term is a variable or has the form $f(s_1, \ldots s_n)$, where $s_1, \ldots s_n$ are terms and f is a function symbol. The letters s, t and u will always denote terms. The size $|s|$ of a term s is 1 if s is a variable and $1 + |s_1| + \ldots + |s_n|$ if $s = f(s_1, \ldots, s_n)$.

An equation has the form s=t. The letter *P* will always range over equations.

An equation system is either the empty equation system 0 or has the

form P_1 &...& P_n, where P_1, ..., P_n are equations. The letter E will always range over equation systems.

A transformation rule for equation systems has the form E -> E', where E and E' are equation systems, meaning the system E is replaced with E'.

The unifications rules:

Tautology
(T) $X=X \; \& \; E \rightarrow E$

Binding
(B) $X=t \; \& \; E \rightarrow X=t \; \& \; [X=t] \; E$

Decomposition
(D) $f(s_1,...,s_n)=f(t_1,...,t_n) \; \& \; E \rightarrow s_1=t_1 \; \&...\& s_n=t_n \; \& \; E$

Orientation
(O) $s=X \; \& \; E \rightarrow X=s \; \& \; E$

In Feature Prolog we distinguish between unbound, bound and type variables. A type variable is a variable whose domain is not the whole universe, but only a subset of it.

A feature term is a common Prolog term or a type variable with a set of features:

<feature term> ::= <common-term> { <list-of-features> };

<common-term> is a Prolog term or a type variable, <list-of-features> is a list of feature declarations. A feature declaration is:

<feature-declaration> ::= <ident> -> <term>;

<ident> is the name of the feature and <term> denotes a Prolog term or a feature term.

In the following we often use the notion of *to unify features,* when we unify the records described by the features; feature value is also a synonym for the record referenced by a feature.

The deduction of Feature Prolog will be done by SLD-resolution, but unification will be changed in the following way:

(1) A variable and a non-variable are unifyable, if the type of the non-variable is a subtype of the variable. Additionally the features of the variable and non-variable have to be unifyable. The variable is bound to the non variable.

(2) Two variables are unifyable, if there is a greatest common subtype of their types and their features are unifyable. The unifier is a new variable, the old variables are bound to.

(3) To decide if a feature Fi of two feature terms is unifyable we distinguish two cases:

If the feature Fi occurs only in one of the two feature terms, then Fi is added with the referenced record to the other feature term.

If the feature Fi occurs in both feature terms, then the features Fi will be unified. If they have a unifier, Fi is copied with the referenced unifier into the instance feature term.

Here is the formal specification for the feature unification rules:

We assume 3 pairwise disjoint, countably infinite sets of symbols:

- Type Symbols: (ϱ, μ, σ)

  There is a subclass of type symbols called featuretype symbols.

- Functions Symbols:

  There are two disjoint subclasses of functions symbols: constructor and feature symbols. All feature symbols are unary functions symbols. The letters *l*, *k* will always range over feature type symbols.

  There are two disjoint subclasses of constructor symbols: Implicit constructor symbols und explicit constructor symbols. E.g. in the signature of Fig.3 *con_person* is an implicit constructor for *person*. Implicite constructors have the prefix *con_*.

- Variables: (X,Y,Z). Each variable has a type ϱX, which is a type symbol; for each type symbol ϱ exists an infinite set of variables with type ϱ. In our examples each item built-up with capitals are variables.

A subtype declaration has the form: $\varrho \leq \mu$.

A function declaration has the form:

$f: \rho_1 \ldots \rho_n \rightarrow \rho$, where n is the arity of f.

To be able to relate features to implicit constructors, we assume a total order " $l \leq k$ " on the set of all feature symbols.

A *signature* Σ is a set of subtype and function declarations. We say f is a constructor of ρ, if f is a constructor and the declarations $f: \rho_1 \ldots \rho_n \rightarrow \mu$ and $\mu \leq \rho$ occur in Σ. l is a feature of ρ, if l is a feature symbol and the declarations $l: \mu \rightarrow \sigma$ and $\rho \leq \mu$ occur in Σ. A Σ-term of type μ is either a variable X with $\mathcal{P}X \leq \mu$, or has the form $f(s_1, \ldots s_n)$, and there are declarations $f: \rho_1 \ldots \rho_n \rightarrow \rho$ and $\rho \leq \mu$ in Σ and s_i are Σ-terms of type ρ_i for all i.

A term of the form $l(X)$ is called a quasi-variable.

A term is called canonical, if it is built only from variables and constructors (explicit or implicit).

For each implicite constructor f with arity n exists n features $l_1, \ldots l_n$, such that $l_i(f(s_1, \ldots s_n)) = s_i$ holds. Hence the implicit constructors are only a syntactically auxiliary to improve the comprehension of the feature unification.

Let con_person be an implicit constructor of type ***person***, with the argument types ***age***, ***status***, ***first_name***, ***person***.

con_person: age, status, first_name, person, person -> person

The features of type ***person*** are unary functions refering to the argument positions of con_person, i.e. we have the equations:

```
period_of_life(con_person(X1, X2, X3, X4, X5)) = X1
social_status(con_person(X1, X2, X3, X4, X5)) = X2
first_id(con_person(X1, X2, X3, X4, X5)) = X3
mother(con_person(X1, X2, X3, X4, X5)) = X4
father(con_person(X1, X2, X3, X4, X5)) = X5
```

Then an element X of type ***person*** has to be of the form con_person(X1,...,X5) and is denoted by the feature term

X:person{period_of_life ->X1:age,...,mother ->X4:person}

Let [*X=t*] be a substitution and *E* an equation system containing the variable X. Then we say that E & X=s can be obtained from [X=s] *E* by unfolding.

An equation s=t is ***trimmed*** if it contains no implicit constructors, t is a canonical term and s is a variable or a quasi variable. A trimmed equation system is a equation system whose equations are all trimmed. By <u>unfolding</u> we can obtain a trimmed equation system.

Have a glance again to the signature in Fig.3 and consider the equation:

```
S=E where S and E are the feature terms
     S:student{first_id ->NAME,
                    mother ->{age ->O:old},
                    major ->BIO:biology}
and
     E:employee{position -> JOB:jobtitle,
                     period_of_life -> AGE:age,
                     salary -> INCOME:income}
```

S is a variable of type ***student*** with the features ***first_id***, ***mother*** and ***major***. Hence S can be represented with implicite constructors as:

```
S = con_student(X1,
                X2,
                NAME,
                con_person (O, X3, X4, X5, X6),
                X7,
                BIO)
```

Similar for the variable E of type ***employee***

```
E = con_employee(AGE, X8, X9, X10, X11, JOB, INCOME)
```

To get a trimmed equation system we unfold the implicit constructor con_student, that is we replace the substructure con_person(O,...) by a new variable ***FOLDING*** of type ***person***.

```
S = con_student(X1, X2, NAME, FOLDING, X7, BIO)
FOLDING = con_person (O, X3, X4, X5, X6)
```

As mentioned before, implicit constructors are only *syntactical sugar* for feature terms, all feature terms can be represented with unary functions. We get the trimmed equation system Γ_example, consisting of common terms only.

> S=E & first_id(S)=NAME & major(S)=BIO & mother(S)=FOLDING &
> period_of_life(FOLDING)=O & period_of_life(E)=AGE &
> position(E)=JOB & salary(E)=INCOME

<u>The feature unifications rules</u>:

<u>Tautology</u>

(T) X=X & Γ -> Γ

<u>Binding</u>

(B) X=t & Γ -> X=t & [X=t]Γ

If X is a variable occurring in Γ but not in t and if Ƥt≤ƤX is.

(B') X=Y & Γ -> X=Z & Y=Z &[X=Z, Y=Z]Γ

If not ƤY≤ƤX and Z is a new variable. ƤZ is the greatest common subtype of ƤX, ƤY.

(B'') *l(X)=Y* & Γ -> Y=Z & *l(X)=Z* &[*l(X)=Z*, Y=Z]Γ

If not ƤY ≤ Ƥl(X) and Z is a new variable. ƤZ is the greatest common subtype of Ƥ *l(X)* und ƤY.

<u>Decomposition</u>

(D) $f(s_1,\ldots,s_n)=f(t_1,\ldots,t_n)$ & Γ -> $s_1=t_1$ &...& $s_n=t_n$ & Γ

<u>Orientation</u>

(O) s=X & Γ -> X=s & Γ

<u>Merging</u>

(M) *l(X)=s* & *l(X)=t* & Γ -> *l(X)=s* & s=t & Σ

Applying the rules to the equation system Γ_example transforms Γ_example to Γ_example'.

1) Rule *B'* to S=E
   S=E & Γ_example -> S=W & E=W & [S=W, E=W]Γ_example
   W is a new variable of type *workstudy* and is bound on S and E. In Γ_example the variables S, W are substituted with E.

2) Rule *M* to period_of_life(W)=MIDDLE and period_of_life(W)=AGE
   period_of_life(W)=MIDDLE & period_of_life(W)=AGE & Γ_example -> period_of_life(W)=AGE & AGE=MIDDLE & Γ_example

3) Rule *O* and *B* to AGE=MIDDLE
   MIDDLE substitutes AGE.

4) Rule *B''* to salery(W)=INCOME
   salary(W)=INCOME & Γ_example ->
   INCOME=LOW_INCOME & salery(W)=LOW_INCOME &
   [salery(W)=LOW_INCOME, INCOME=LOW_INCOME]Γ_example
   LOW_INCOME is a new variable of type *low_income* , the variable LOW_INCOME substitudes the variable INCOME in Γ_example.

After applying the rules O and B to Γ_example we get the solved equation system Γ_example':

```
S=W & E=W & INCOME=LOW_INCOME
with W:workstudy{period_of_life -> MIDDLE,
                 first_id -> NAME,
                 mother -> FOLDING,
                 major -> BIO,
                 position -> JOB,
                 salary -> LOW_INCOME}
```

The example above is simple in the sense that it does not illustrate the use of inheritance among complex feature terms. In the next example the type *person* has the features id-first, period_of_life (see the *has-feature* declarations). Now let us suppose that we also want to specify that a *student* is of subtype *person,* i.e. that it inherits whatever attributes are imposed on *person*. Additional *student* can achieve some new attributes, e.g. every student has a major which must be a subject, and further constrains the value of feature *period_of_life* with *middle_age*. This is achieved by:

```
has-feature(student,major,course).
has_feature(student,period_of_life,middle_age).
```

For the ordering of the feature types see Fig3.

Example 2

```
% THE DECLARATION PART: %
    subsort(..,..) see FIG.3
    csort(..,..) see FIG.3
    has-feature(person,period_of_life,age).
    has-feature(person,social-status,status).
    has-feature(person,first_id,first_name).
    has-feature(person,mother,person).
    has-feature(person,father,person).
    has-feature(student,major,subject).
    has-feature(student,period_of_life,middle).
    has-feature(employee,position,jobtitle).
    has-feature(employee,salary,income).
    has-feature(workstudy,salary,low_income).
    has-feature(s1,social_status,local).
    has-feature(s2,social_status,eg_member).
    has_feature(w1,social_status,local).
    has_feature(w2,social_status,no_eg_member).
    has-feature(s1,major,biology).
    has-feature(s2,major,chemestry).
    has-feature(w1,major,physical_education).
    has-feature(w2,major,computer_science).
    has-feature(s1,first_id,peter).
    has-feature(s2,first_id,peter).
    has_feature(w1,first_id,mary).
    has_feature(w2,first_id,abdula).
```

To express that a foreign person has the *social_status no_eg_member*.

```
% THE FACT FOR foreign %
    foreign(X:person {social_status ->Y:no_eg_member}).
```

To formulate that less than 4 lectures are only a few, we define it by the length of the *course_enrollment_list*.

```
% THE FACTS FOR few_course_enrollment %
    few_courses_enrollment(no_lecture).
    few_courses_enrollment([X:lecture,no_lecture]).
    few_courses_enrollment([X:lecture,Y:lecture,no_curriculum]).
    few_courses_enrollment
        ([X:lecture,Y:lecture,Z:lecture,no_lecture]).
```

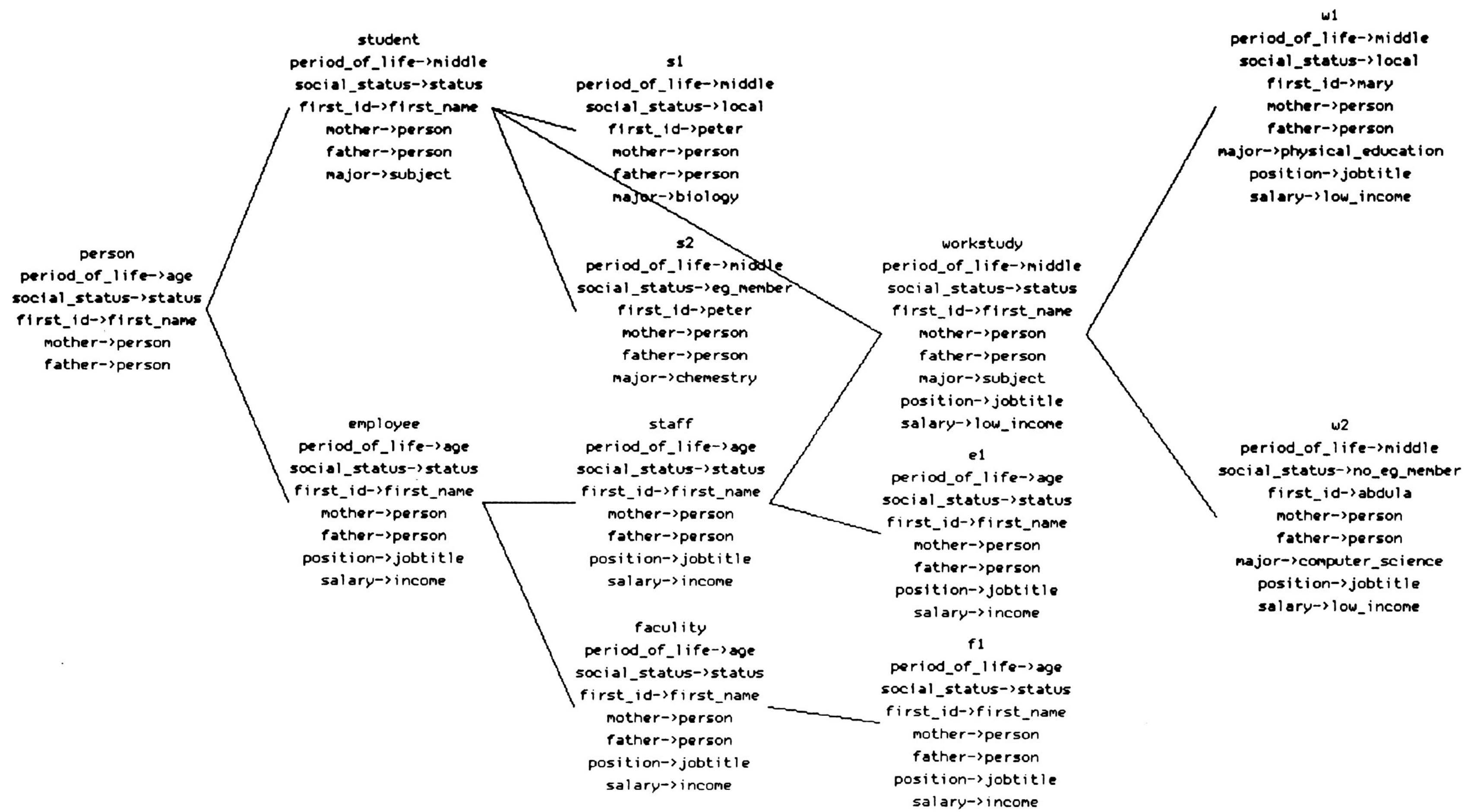


FIG.3 The type hierarchy

The relation of a student and his course enrollment is given in the facts for ***takes***.

```
% THE FACTS FOR takes %
    takes(s1, [chem1,chem2,bio2,ma3,no_lecture]).
    takes(s2, [chem1,chem2,bio3,ma3,no_lecture]).
    takes(w1,[ma1,ma2,ma3,pe2,no_lecture]).
    takes(w2,[ma1,comp2,no_lecture]).
```

Finally we define that all students taking less than 3 lectures are considered parttime students.

```
% THE RULE FOR partime: %
part_time(X:student):-
    takes(X:student,CL:course_list)
    few_courses_enrollment(CL:course_list).
```

```
% THE QUERY: %
    ?- foreign(Y:workstudy{salary   -> Z:low_income,
                           first_id -> X:first_name}),
       part_time(Y).
```

The answer of the query will be ***abdula***, because he is a foreign employee, who is also parttime student and his salary class is in the low income class.

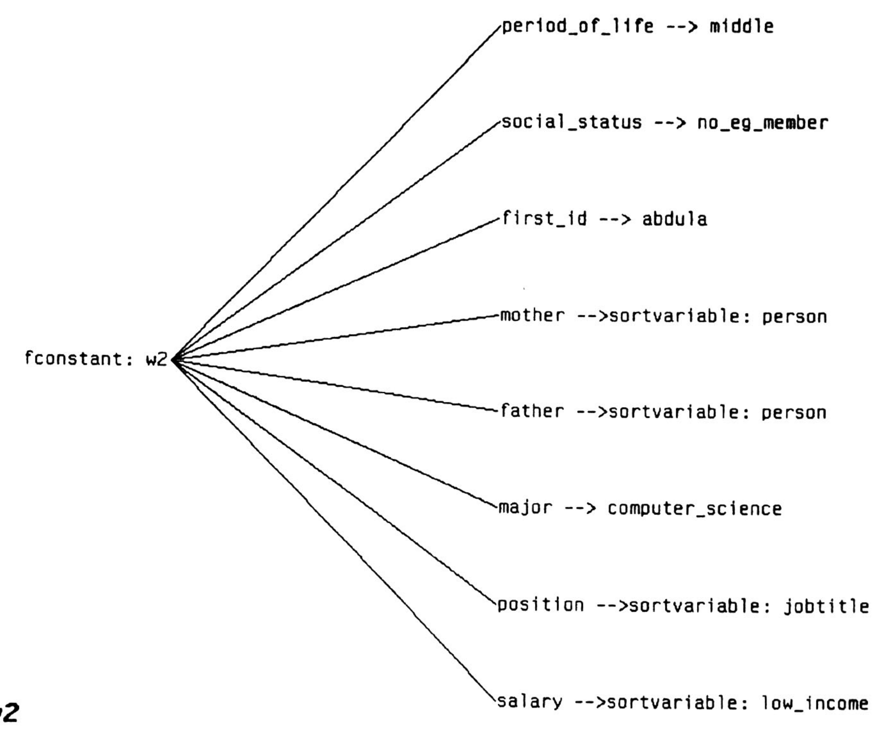


Fig.4 The object *w2*

Constraint Propagation with Feature Prolog

In Feature ProLog constraint propagation supports the reduction of the search space.

Example:

The signature is the same as in Fig.3 but the feature type ***person*** is extended with the feature ***room-mate***. ***Room-mate*** of ***person*** is of type ***person***, ***room-mate*** for student is constrained to ***student*** or a subtype of ***student***. The equation system is:

```
S=E with
    E:employee{room-mate->
                  person{room-mate->
                           person{room-mate->person}}}
```

and S is of type ***student***.

During unification the constraint ***room-mate*** -> student is propagated to all room-mates of E. After applying rule *B'* to S=E the new variable W of type ***workstudy*** is bound to S and E. The solved equation system:

```
S=W & E=W with
W:workstudy{room-mate->
               student{room-mate->
                         student{room-mate->student}}}
```

3.An Instruction Set for Feature Prolog

In this section we define the feature unification instructions, extending the WARREN Abstract Machine (WAM) to features. We use the same notion as in /Be 85/.

The WAM is defined by an abstract instruction set for the compilation of Prolog programs. One idea behind it is to transform the unification procedure for two unknown terms into several special unification procedures determined by the structures of the clause headers. Those structures are completely specified when the program is created. Hence they can be compiled into special unification code, that can only unify the headers with terms with analoguous structure, but can do this very efficiently.

The architecture of the WAM is very similar to the architecture of a conventional computer. The WAM has two different memories, one for the program and one for the data. The data are stored onto the *GLOBAL STACK*.

If we compile Prolog in a virtuell code then we are confronted with three problems:

1) Each goal in Prolog is considered as a procedure call, therefore an activation record is pushed onto the runtime stack. The task of the compiler is to compute the size of the activation record and to generate the according code. The management of the runtime system of the WAM is very similar to conventional runtime systems. Warren calls the runtime stack *LOCAL STACK*.
2) New in Prolog is the *TRAIL STACK*, where the old bindings of variables have to be stored. During backtracking the variables get the bindings they had until the old choice point.

3) If Prolog is interpreted, a complete unification is executed for each argument of the clause head at a call of this clause. The Prolog compiler generates special code before runtime in order to substitute complete unification with the really necessary part of the unification. E.g. if the first argument of a clause head is a variable, an instruction is generated, that binds the variable of the clause head to the first argument of the calling goal. That means that there is executed only a write operation - no read operation in contrast to the complete unification. The class of instructions operating on clause headers are called *GET_INSTRUCTIONS*. All Warren instructions operating on clause head arguments start with the prefix *GET_*. For the arguments of a goal the compiler generates code to built-up these arguments very efficiently during runtime. Each instruction of this class has the prefix *PUT_*. If structures occur in a literal, the compiler generates code to require a special access mechanism for the arguments. The instructions belonging to this class start with the prefix *UNIFY_*.

A clause with the head h and the two goals g1 and g2:

h(...) :- g1(...), g2(...) is transformated into code doing the following operations:

1. allocate environment
2. unify h(...) with the calling goal
3. initialize argument registers for g1
4. call g1
5. initialize argument registers for g2
6. call g2
7. deallocate environment
8. return from clause

The first step is to allocate an activation record onto the *LOCAL STACK*. In step 2 the arguments of h(...) have to be unified with

the arguments of the calling clause - the compiler generates code consisting of GET_... and UNIFY_ instructions. In step 3 the arguments for the first goal (g1) are built-up with PUT_... and UNIFY_... instructions. In the next step a jump instruction is executed for calling g1. After g1 succeeds, the code builts-up the arguments for g2 (step 6). After calling g2 and a successful execution of g2 the activation record is deallocated (step 7) and control is given back to the calling clause (step 8).

The reader is referred to /War 77/, /War 83/ and the WAM tutorial /GLLO 84/ for a more detailled description. In respect of inherited hierarchies we need some new GET, PUT and UNIFY instructions. We use a similar pseudo code as /Be 85/, for our following definitions of these new instructions. A first step to implement the feature instructions was the implementation of sort instructions, described in /Bue 85/, /Hub 85/, /Var 87/. This approach extends Prolog with types ordered by subtyping, but still not structured by features. In the following we often use the synonym *sort* for type - the reason is that the feature instructions are built-up from the sort instructions. The information about the types is stored in a type table, the information about feature constants is stored in a constant table. If we need the value of a feature Fi of a constant C, we can get it very fast using the constant table, which gives us the reference of the feature value on the *GLOBAL STACK*. To be able to relate features to implicit constructors, we assume a total order of all features. During compilation the features are converted into integers. The total order relation is given with the *less_than* relation for the internal representation of the features.

In the following we use the notion feature constant, feature variable,

feature structure for terms corresponding with a set of features.

PUT_Y_FVARIABLE Yn,Ai,S,Arity

This instruction represents a goal argument that is an unbound, permanent, feature variable. It puts the adress of variable Yn into register Ai. An entry of 'REF GPOS' is made in the *LOCAL STACK*. In the *GLOBAL STACK* at position GPOS the variable is determined by an offset, containing the tag FEATURE_VARIABLE and the type S. Arity represents the number of features for the variable, it is used to compute the next free memory cell in the *GLOBAL STACK*.

```
Tag.MEM(Ai) <-- REF
MEM(Ai) <-- GPOS
Tag.MEM(Currenv + n) <-- REF
MEM(Currenv + n) <-- GPOS
TAG.MEM(GPOS) <-- FEATURE_VARIABLE
MEM(GPOS) <-- GPOS + n
GPOS <-- GPOS + 1
TAG.MEM(GPOS) <-- S
GPOS <-- GPOS + 1
Next-free-memory-cell <-- Arity + GPOS
Current-sort <-- S
MODE <-- WRITE
```

PUT_FCONSTANT C,Ai

This instruction represents a goal argument that is a feature constant. As briefly described before, feature constants are stored permanently in the *GLOBAL STACK*, this is done before runtime. With the fast access function Get_Const_Ref we get the reference of the constant and put it into register Ai.

ATTENTION! The mode will be switched from WRITE to CMERGE, since all features of the feature constant have to be unified with the stored features of the constant. The advantage of this modification is that the search space will be reduced (see INSTANT_INSTRUCTIONS).

```
TAG.Ai <-- REF
Ai <-- Get-Const-Ref(C)
MODE <-- CMERGE
Nextarg <-- C
```

PUT_FSTRUCTURE An,Ai,S,Arity

This instruction represents a goal argument that is a feature structure. The structure itself was created on the *GLOBAL STACK* before runtime.

```
Ai <-- GPOS
Tag.Ai <-- REF
Tag.MEM(GPOS) <-- FEATURE_STRUCTURE
MEM(GPOS) <-- GPOS +1
GPOS <-- GPOS + 1
Tag.MEM(GPOS) <-- REF
MEM(GPOS) <-- An
GPOS <-- GPOS + 1
Next-free-memory-cell <-- GPOS + Arity
Current-sort <-- S
MODE <-- WRITE
```

If we deal with feature terms occurring in the clause head we have the problem to allocate the *GLOBAL STACK* with the number of all features belongs to the corresponding feature type. Therefore the function *Get-nr-of-features* with the argument *Sort* computes the number of all features for the feature type *Sort*. If unification of a feature term with an unbound - or sort variable is done, we can optimize the allocation process. Hence we allocate the *GLOBAL STACK* only with the number of the features occurring in the feature term. In our instruction extension the argument 'Arity gets the information about the number of features of a feature term.

We have two additional MODES: VMERGE and CMERGE.

CMERGE is used during unification of feature constants, VMERGE is used during unification of feature variables and feature structures. For a detailed explanation see the UNIFY_FEATURE... instruction section. Because of the feature extension we need the new registers *Current-sort* and *Next-free-memory-cell*.

Current-sort stores the current sort of a feature term. We need Current-sort to compute the constraints of the features during unification. Next-free-memory-cell is a second stack pointer to the *GLOBAL STACK*. After the allocation of the *GLOBAL STACK* with the number of features of a feature term the stackpointer GPOS refers to the first memory cell for the feature of a feature term. The terms denoted

by the features are stored in the memory cells referred by Next-free-memory-cell.

GET_Y_FVARIABLE Yn,Ai,S,Arity

This instruction marks the beginning of a feature variable occurring as a head argument. The instruction gets the value of register Ai and dereferences it. If the result is a reference to an unbound variable or sort variable the variable is bound to the feature variable. If the result is a feature variable a new feature variable is created onto the *GLOBAL STACK* with the common subsort of both sorts and execution proceeds in VMERGE mode. If the result is a feature constant then the sort of the feature constant has to be a subsort of the feature variable and execution proceeds in CMERGE mode. The common subsort of both sorts is written in register Current-subsort. Register Next-free-memory-cell refers to the next free memory cell after the allocation of the *GLOBAL STACK* with the number of features of the feature term.

```
While Tag.Ai = REF DO Ai <-- MEM(Ai)
CASE Tag.Ai
     UNBOUND:
          Current-sort <-- S
          TRAIL(AI)
          Tag.MEM(GPOS) <-- FEATURE_VARIABLE
          MEM(GPOS) <-- GPOS + 1
          Tag.MEM(Currenv + n) <-- REF
          MEM(Currenv + n) <-- GPOS
          Tag.MEM(Ai) <--REF
          MEM(Ai) <-- GPOS
          GPOS <-- GPOS + 1
          Tag.MEM(GPOS) <-- Current-sort
          GPOS <-- GPOS + 1
          MODE <-- WRITE
          Next-free-memory-cell <-- GPOS + Arity
     F_VARIABLE:
          Current-sort                <--
          Get-common-subsort(S,Tag.MEM(MEM(Ai)))
          Current-sort <--
               Get-common-subsort(S,Tag.MEM(MEM(Ai)))
          IF Current-sort THEN
               TRAIL(AI)
               MEM(Ai) <-- GPOS
               MEM(Currenv + n) <-- GPOS
               Tag.MEM(Currenv + n) <-- REF
               Tag.MEM(GPOS) <-- Current-sort
               Nextarg <-- Ai + 1
               MEM(GPOS) <-- GPOS
               GPOS <-- GPOS + 1
               MODE <-- VMERGE
               Next-free-memory-cell <-- GPOS +
                    (Get-nr-of-features Current-sort)
          ELSE FAIL
```

```
FEATURE_CONSTANT:
     IF Is-subsort (Tag.MEM(MEM(Ai)),S) THEN
          MEM(Currenv + n) <-- MEM(Ai)
          Tag.MEM(Currenv + n) <-- REF
          Nextarg <-- Tag.MEM(MEM(Ai))
          MODE <-- CMERGE
     ELSE FAIL
S_STRUCTURE:
     If Is-subsort (Tag.MEM(MEM(Ai)),S) THEN
          Current-sort <-- Tag.MEM(MEM(Ai))
          MEM(Currenv + n) <-- MEM(Ai)
          Tag.MEM(Currenv + n) <-- REF
          MODE <-- VMERGE
     ELSE FAIL
FEATURE_STRUCTURE:
     If Is-subsort (Tag.MEM(MEM(Ai)),S) THEN
          Current-sort <-- Tag.MEM(MEM(Ai))
          MEM(Currenv + n) <-- MEM(Ai)
          Tag.MEM(Currenv + n) <-- REF
          Nextarg <-- Tag.MEM(MEM(Ai))
          MODE <-- VMERGE
     ELSE FAIL
OTHERWISE:
     IF Is-sort-tag(Tag.MEM(Ai)) THEN
          Current-sort <--
               Get-common-subsort(S,Tag.MEM(Ai))
          IF Current-sort THEN
               TRAIL(AI)
               Tag.MEM(GPOS) <-- FEATURE_VARIABLE
               MEM(GPOS) <-- GPOS + 1
               MEM(Ai) <-- GPOS
               Tag.MEM(Ai) <-- REF
               MEM(Currenv + n) <-- GPOS
               Tag.MEM(Currenv + n) <-- REF
               GPOS <-- GPOS + 1
               Tag.MEM(GPOS) <-- Current-sort
               GPOS <-- GPOS + 1
               MODE <-- WRITE
               Next-free-memory-cell <-- GPOS + Arity
          ELSE FAIL
     ELSE FAIL
```

The GET_X_FVARIABLE and GET_FVARIABLE_VOID instructions are the same as GET_Y_FVARIABLE, but (Currenv + n) is replaced by Ai or is needless.

GET_FCONSTANT C, Ai

This instruction represents a feature constant occurring as a head argument. The instruction gets the value of register Ai and dereferences it. If the result is a reference to an unbound variable or sort variable the variable is bound to the feature constant. Is the result a feature variable a sort check is executed and all features of the variable will be unified with the features of the constant. Remember that a feature constant is stored permanently onto the *GLOBAL STACK*. Therefore we get the reference to the feature value with 'Get-feature-of-constant C Fi'.

```
While Tag.Ai = REF DO Ai <-- MEM(Ai)
CASE Tag.Ai
     UNBOUND:
          TRAIL(AI)
          MEM(Ai) <-- Get-const-ref(C)
          Tag.MEM(Ai) <-- REF
     F_VARIABLE:
          IF Is-subsort(C,Tag.MEM(MEM(Ai))) THEN
               TRAIL(AI)
               Tag.MEM(Ai) <-- REF
               MEM(Ai) <-- Get-const-ref(C)
               repeat
                    Ai <-- Ai + 1
                    UNIFY(MEM(MEM(Ai)),
                         Get-feature-of-constant(
                         C,
                         Tag.MEM(MEM(Ai)))
               UNTIL Tag.MEM(AI)=END_OF_FEATURE_TERM
          ELSE FAIL
     FEATURE_CONSTANT:
          IF S<>Tag.MEM(MEM(Ai)) THEN
               FAIL
     OTHERWISE:
          IF Is-sort-tag(Tag.MEM(Ai) THEN
               Sort <-- Is-subsort(C,Tag.MEM(Ai))
               IF Sort THEN
                    TRAIL(AI)
                    MEM(Ai) <-- Get-const-ref(C)
                    Tag.MEM(Ai) <-- REF
               ELSE FAIL
          ELSE FAIL
```

GET_FSTRUCTURE An,Ai,S,Arity

This instruction marks the beginning of a feature structure occurring as a head argument. The instruction gets the value of register Ai and dereferences it. If the result is a reference to an unbound variable or sort variable, this variable is bound to the feature structure. If the result is a feature variable, sort structure or a feature structure then a new feature structure is created onto the *GLOBAL STACK* and execution proceeds in VMERGE mode. The sort of the structure is written in register Current-subsort. Register Next-free-memory-cell refers to the next free memory cell after the allocation of the *GLOBAL STACK* with the number of features of the feature term. Note that the unification

of a feature structure is executed in two steps (it is the *unfolding* technique described before). The first step is to unify all features of the feature structure. The second step is to unify the structure themselve. The GET_FSTRUCTURE instruction prepares the register An and a memory cell for the second step. We will demonstrate it through the following example:

```
p(foo(a,b){Fi -> A:s1, Fj -> B:s2}) :- ...
```

The compiler generates (in the next version) the following code:

```
GET_FSTRUCTURE(10, 0, sort_of_foo, 2)
UNIFY_FEATURE_Y_SVARIABLE (Fi, 11, s1)
UNIFY_LAST_FEATURE_Y_VARIABLE (Fj, 12, s2)
GET_S_STRUCTURE (foo, sort_of_foo, 10)
UNIFY_CONSTANT (a)
UNIFY_CONSTANT (b)
     While Tag.Ai = REF DO Ai <-- MEM(Ai)
     CASE Tag.Ai
          UNBOUND:
               Current-sort <-- S
               TRAIL(AI)
               Tag.MEM(GPOS) <-- FEATURE_STRUCTURE
               MEM(GPOS) <-- GPOS + 1
               Tag.MEM(An) <-- REF
               MEM(An) <-- GPOS + 1
               Tag.MEM(Ai) <--REF
               MEM(Ai) <-- GPOS
               GPOS <-- GPOS + 1
               Tag.MEM(GPOS) <-- Current-sort
               MEM(GPOS) <-- GPOS
               GPOS <-- GPOS + 1
               MODE <-- WRITE
               Next-free-memory-cell <-- GPOS + Arity
          F_VARIABLE:
               Current-sort <-- Is-subsort(S,Tag.MEM(MEM(Ai)))
               IF Current-sort THEN
                    TRAIL(AI)
                    Tag.MEM(GPOS) <-- FEATURE_STRUCTURE
                    MEM(GPOS) <-- GPOS + 1
                    Tag.MEM(An) <-- REF
                    MEM(An) <-- GPOS + 1
                    Tag.MEM(Ai) <-- REF
                    MEM(Ai) <-- GPOS
                    GPOS <-- GPOS + 1
                    Tag.MEM(GPOS) <-- Current-sort
                    MEM(GPOS) <-- GPOS + 1
                    Nextarg <-- Ai + 1
                    GPOS <-- GPOS + 1
                    MODE <-- VMERGE
                    Next-free-memory-cell <-- GPOS +
                         (Get-nr-of-features Current-sort)
               ELSE FAIL
```

```
S_STRUCTURE:
     Current-sort <-- Is-subsort(S,Tag.MEM(MEM(Ai)))
     If Current-sort THEN
          TRAIL(AI)
          Tag.MEM(GPOS) <-- FEATURE_STRUCTURE
          MEM(GPOS) <-- GPOS + 1
          Tag.MEM(An) <-- REF
          MEM(An) <-- GPOS + 1
          GPOS <-- GPOS + 1
          Tag.MEM(GPOS) <-- REF
          MEM(GPOS) <-- MEM(MEM(AI))
          Tag.MEM(Ai) <-- REF
          MEM(Ai) <-- GPOS - 1
          GPOS <-- GPOS + 1
          MODE <-- WRITE
          Next-free-memory-cell <-- GPOS +
               (Get-nr-of-features Current-sort)
     ELSE FAIL
FEATURE_STRUCTURE:
     Current-sort <-- Is-subsort(S,Tag.MEM(MEM(Ai)))
     If Current-sort THEN
          TRAIL(AI)
          Tag.MEM(GPOS) <-- FEATURE_STRUCTURE
          MEM(GPOS) <-- GPOS + 1
          Tag.MEM(An) <-- REF
          MEM(An) <-- GPOS + 1
          GPOS <-- GPOS + 1
          Tag.MEM(GPOS) <-- REF
          MEM(GPOS) <-- MEM(MEM(AI))
          Tag.MEM(Ai) <-- REF
          MEM(Ai) <-- GPOS - 1
          Nextarg <-- Ai + 1
          GPOS <-- GPOS + 1
          MODE <-- VMERGE
          Next-free-memory-cell <-- GPOS +
               (Get-nr-of-features Current-sort)
     ELSE FAIL
OTHERWISE:
     IF Is-sort-tag(Tag.MEM(Ai)) THEN
          Current-sort <-- Is-subsort(S,Tag.MEM(Ai))
          IF Current-sort THEN
               TRAIL(AI)
               Tag.MEM(GPOS) <-- FEATURE_STRUCTURE
               MEM(GPOS) <-- GPOS + 1
               MEM(Ai) <-- GPOS
               Tag.MEM(Ai) <-- REF
               MEM(An) <-- GPOS + 1
               Tag.MEM(An) <-- REF
               GPOS <-- GPOS + 1
               Tag.MEM(GPOS) <-- UNBOUND
               MEM(GPOS) <-- GPOS
               GPOS <-- GPOS + 1
               MODE <-- WRITE
               Next-free-memory-cell <-- GPOS + Arity
          ELSE FAIL
     ELSE FAIL
```

The following instructions are the instructions for the feature term arguments. We can split them into two disjoint classes:

Instructions for feature variables and feature structures begin with the prefix 'UNIFY_FEATURE'.

Instructions for feature constants begin with the prefix 'INIT_CONST'.

We use a different instruction set for feature constants as for feature structures and variables because feature constants have a fixed storage in the memory. If we define a term referenced by a feature of a constant we have to unify it with the feature value in the fixed storage.

Example:

> Assume we have the feature constant *peter* with the feature *mother* that is bound to a sort variable of type *person* before runtime. During runtime *anne* is bound to *peter's mother*. If we have a rule like
>
> p(X) :- q(peter {mother -> mary}).
>
> and call *p* then a failure has to occur, while binding *mary* to *peter's mother*.

The subroutine 'Copy-features Nextarg Fi' copies all features less than Fi onto the *GLOBAL STACK*. Before the features are copied a sort check is executed and if necessary the sort is altered. The value of *Copy-features* is the first feature > Fi, if the sort check succeeds.

The subroutine 'Get-sort-of-feature Current-sort Fi' computes the sort of the feature *Fi* of feature type *Current-sort*. The reference to the feature value of a feature of a constant is obtained by the subroutine 'Get-feature-of-constant Nextarg Fi'.

The next problem is to generate the *end* mark of a feature term. We introduce so-called UNIFY_FEATURE_..._LAST instructions. They do the same as the UNIFY_FEATURE_... instructions but after the last feature the flag 'END_OF_FEATURE_TERM is pushed onto the *GLOBAL STACK*. 'Is-sort-compatibel Fi Sort' dereferences the feature Fi of a feature term and executes a sort check with the reference adress. If necessary the sort will be altered. the value of 'Is-sort-compatibel is true if the sorts are compatibel else nil.

UNIFY_FEATURE_Y_VARIABLE Fi,Yn

This instruction represents a feature term argument that is a permanent, unbound variable. If the instruction is in WRITE mode, it pushes a new variable onto the *GLOBAL STACK*, and links the feature Fi with that variable. The reference to the variable is stored onto the *LOCAL STACK* at position (Currenv + n). The sort of the variable computes the function *Get-sort-of-feature* with the arguments Current-sort and Fi. If the instruction is in VMERGE mode, it simply gets the next feature from Nextarg. If the intern feature name is less than Fi (note: we have a '<' order onto the features) then we copy the feature, referenced by Nextarg onto the *GLOBAL STACK* and increment Nextarg. The same operation is repeated until one of the following termination conditions succeeds:

If the feature referenced by Nextarg is greater than Fi we do the same as in WRITE mode. If the intern feature name is equal to Fi, we copy the feature onto the *GLOBAL STACK* and puts in MEM(Currenv + n) the reference GPOS.

Is the instruction in CMERGE mode, we get the reference to the feature Fi in the constant with the subroutine 'Get-feature-of-constant Nextarg Fi' and store the reference of Fi in the *LOCAL STACK* at position: (Currenv + n)

```
CASE MODE
     WRITE:
          Tag.MEM(GPOS) <-- Fi
          MEM(GPOS) <-- Next-free-memory-cell
          GPOS <-- GPOS + 1
          Tag.MEM(Currenv + n) <-- REF
          MEM(Currenv + n) <-- Next-free-memory-cell
          MEM(Next-free-memory-cell) <--
               Next-free-memory-cell
          Tag.MEM(Next-free-memory-cell) <--
               Get-sort-of-feature(Current-sort, Fi)
          Next-free-memory-cell <--
               Next-free-memory-cell + 1
```

```
VMERGE:
     WHILE (< Tag.MEM(Nextarg) Fi)
          Copy-feature(Nextarg, Fi)
     IF Tag.MEM(Nextarg) = Fi THEN
          Tag.MEM(GPOS) <-- Fi
          MEM(GPOS) <-- MEM(MEM(Nextarg))
          GPOS <-- GPOS + 1
          Tag.MEM(Currenv + n) <-- REF
          MEM(Currenv + n) <-- MEM(MEM(Nextarg))
          Nextarg <-- Nextarg + 1
     ELSE
          Tag.MEM(GPOS) <-- Fi
          MEM(GPOS) <-- Next-free-memory-cell
          GPOS <-- GPOS + 1
          Tag.MEM(Currenv + n) <-- REF
          MEM(Currenv + n) <-- Next-free-memory-cell
          MEM(Next-free-memory-cell) <--
               Next-free-memory-cell
          Tag.MEM(Next-free-memory-cell) <--
               Get-sort-of-feature(Current-sort, Fi)
          Next-free-memory-cell <--
               Next-free-memory-cell + 1
CMERGE:
     Tag.MEM(Currenv + n) <-- REF
     MEM(Currenv + n) <--
          MEM(Get-feature-of-constant(Nextarg,Fi))
```

UNIFY_FEATURE_Y_SVARIABLE Fi,Yn,S

This instruction represents a feature term argument that is a permanent, sort variable. If the instruction is in WRITE mode, it pushes a new sort variable onto the *GLOBAL STACK*, and links the feature Fi with that sort variable. The reference is stored onto the *LOCAL STACK* at position (Currenv + n). If S is a subsort of 'Get-sort-of-feature Current-sort Fi' then the new variable has the sort S else the sort of the variable is the sort of the feature Fi of Current-sort. If the instruction is in VMERGE mode it simply gets the next feature from Nextarg. If the intern feature name is less than (note: we have a '<' order onto the features) we copy the content of Fi onto the *GLOBAL STACK* and increment Nextarg. The same operation is repeated until one of the following termination conditions succeeds:

If the feature referenced by Nextarg is greater than Fi we do the same as in WRITE mode. If the features are equal, a sort check will be executed and if the sorts are compatibel, we copy the value referenced by Nextarg onto the *GLOBAL STACK* and increment Nextarg. Otherwise FAIL is executed.

Is the instruction in CMERGE mode, we get the reference to the feature Fi in the constant with 'Get-feature-of-constant Nextarg Fi', executes a sort check and stores the reference in the *LOCAL STACK* at position: (Currenv + n).

```
If Is-subsort(S, Get-sort-of-feature(Current-sort, Fi)) THEN
     S
ELSE
     S <-- Get-sort-of-feature(Current-sort, Fi))
CASE MODE
     WRITE:
          Tag.MEM(GPOS) <-- Fi
          MEM(GPOS) <-- Next-free-memory-cell
          GPOS <-- GPOS + 1
          Tag.MEM(Currenv + n) <-- REF
          MEM(Currenv + n) <-- Next-free-memory-cell
          MEM(Next-free-memory-cell) <--
               Next-free-memory-cell
          Tag.MEM(Next-free-memory-cell) <-- S
          Next-free-memory-cell <--
               Next-free-memory-cell + 1
     VMERGE:
          WHILE (< Tag.MEM(Nextarg) Fi)
               Copy-feature(Nextarg,Fi)
          IF Tag.MEM(Nextarg) = Fi THEN
               If Is-sort-compatibel(MEM(Nextarg),S) THEN
                    Tag.MEM(Currenv + n) <-- REF
                    MEM(Currenv  +  n)  <--  MEM(Nextarg)
                    Tag.MEM(GPOS) <-- Fi
                    Tag.MEM(GPOS) <-- MEM(Nextarg)
                    GPOS <-- GPOS + 1
               ELSE FAIL
          ELSE
               Tag.MEM(GPOS) <-- Fi
               MEM(GPOS) <-- Next-free-memory-cell
               GPOS <-- GPOS + 1
               Tag.MEM(Currenv + n) <-- REF
               MEM(Currenv + n) <-- Next-free-memory-cell
               MEM(Next-free-memory-cell) <--
                    Next-free-memory-cell
               Tag.MEM(Next-free-memory-cell) <-- S
               Next-free-memory-cell <--
                    Next-free-memory-cell + 1
     CMERGE:
          Feature-arg <--Get-feature-of-constant(Nextarg,Fi)
          IF Is-sort-compatibel(MEM(Featurearg),S) THEN
               Tag.MEM(Currenv + n) <-- REF
               MEM(Currenv + n) <-- MEM(Featurearg)
          ELSE FAIL
```

INIT_CONST_FCONST Fi,C,Ci

This instruction represents a feature constant argument that is a constant. Since the feature constants are defined before runtime, every feature of C defined in a rule has to be unified with the stored feature value of C. If the feature argument Fi of C is an unbound variable the unbound variable is bound to the constant Ci. If the feature argument is a sort variable, the constant Ci has to be a subsort of the sort variable and the sort variable is bound to Ci. In the case that the feature argument Fi is a constant the name of the constant must be identical to Ci. If the feature argument is a feature variable, then the feature variable is bound to Ci. Ci has to be a subsort of the sort of the feature variable and all feature arguments of the variable have to be unifyable with the feature arguments of Ci.

```
Featurearg <-- Get-feature-of-constant(C,Fi)
CASE Tag.Featurearg
     UNBOUND:
          TRAIL(Featurearg)
          Tag.MEM(Featurearg) <-- FEATURE_CONSTANT
          MEM(Featurearg) <-- Get-const-ref(C)
     F_VARIABLE:
          IF Is-subsort(Ci,Tag.MEM(MEM(featurearg))) THEN
               Ref-copy          <--           MEM(Featurearg)
               TRAIL(MEM(Featurearg))
               Tag.MEM(MEM(Featurearg)) <-- FEATURE_CONSTANT
               MEM(MEM(Featurearg)) <-- Get-const-ref(Ci)
               REPEAT
                    Ref-copy <-- Ref-copy + 1
                    UNIFY(MEM(MEM(Feature-arg)),
                         MEM(Get-feature-of-constant
                              Ci
                              Tag.MEM(Featurearg)))
               UNTIL Tag.MEM(Ref-copy) = END_OF_FEATURE_TERM
               ELSE FAIL
     FEATURE_CONSTANT:
          IF Ci <> Tag.(MEM(MEM(Featurearg))) THEN
               FAIL
     OTHERWISE:
          IF Is-sort-tag(Tag.MEM(MEM(Featurearg)) THEN
               sort <-- .
                    Is-subsort(Ci,Tag.MEM(MEM(Featurearg)))
               IF sort THEN
                    TRAIL(MEM(Featurearg))
                    MEM(MEM(Featurearg)) <-- .
                         Get-const-ref(Ci)
                    Tag.MEM(MEM(Featurearg)) <-- .
                         FEATURE_CONSTANT
               ELSE FAIL
          ELSE FAIL
```

INIT_FCONST_Y_SVARIABLE Fi,C,S,Yn

This instruction represents a feature constant argument of a feature constant that is a sort variable. We distinguish between 5 possibilities.

1. If the feature argument Fi of C is an unbound variable, the unbound variable is bound to the sort variable.
2. If the feature argument is a sort variable, a sort check is executed and if necessary the sort of the variable is altered.
3. If the feature argument is a constant, the constant has to be a subsort of S.
4. If the feature argument is a feature variable, the sort variable is bound to the feature variable - if there exists a greatest common subsort.
5. If the feature argument is a structure, the sort of the structure has to be a subsort of S.

```
Featurearg <-- Get-feature-of-constant(C,Fi)
CASE Tag.MEM(Featurearg)
     UNBOUND:
          TRAIL(MEM(Featurearg))
          Tag.MEM(MEM(Featurearg)) <-- S
     F_VARIABLE:
          Sort <--
               Get-common-subsort(
               S,
               Tag.MEM(MEM(Featurearg)))
          IF Sort THEN
               TRAIL(MEM(MEM(Featurearg)))
               Tag.MEM(MEM(Featurearg)) <-- Sort
               ELSE FAIL
     S_STRUCTURE:
          IF (not(Is-subsort(Tag.(MEM(MEM(Featurearg))),S))
          THEN
               FAIL
     FEATURE_CONSTANT:
          IF (not(Is-subsort(Tag.(MEM(MEM(Featurearg))),S))
          THEN
               FAIL
     FEATURE_STRUCTURE:
          IF (not(Is-subsort(Tag.(MEM(MEM(Featurearg))),S))
          THEN
               FAIL
     OTHERWISE:
          IF Is-sort-tag(Tag.MEM(MEM(Featurearg)) THEN
               Sort <--
                    Get-common-subsort(
                    S,
                    Tag.MEM(MEM(Featurearg)))
               IF Sort THEN
                    TRAIL(MEM(Featurearg))
                    Tag.MEM(MEM(Featurearg)) <-- Sort
               ELSE FAIL
          ELSE FAIL
```

The INIT_CONST_X_VARIABLE and INIT_CONST_Y_VARIABLE instructions put the address of the feature argument Fi of the constant C onto the *LOCAL STACK* (currenv + n) or in Register Ai.

INIT_CONST_Y_VALUE Fi,C,Yn

This instruction represents a feature constant argument that is a variable bound to some global value. It gets the argument of the constant with 'Get-feature-of-constant C,Fi' and unifies it with the value of the variable Yn.

```
(UNIFY
     MEM(Currenv + n),
     MEM(Get-feature-of-constant C,Fi))
```

The INIT_CONST_X_VALUE instruction is the same as the INIT_CONST_Y_VALUE instruction, but (Currenv + n) is replaced by register Ai.

Conclusion

Feature unification is an elegant method to solve constraints with an efficient inheritance method without the use of formal deduction steps - computation instead of searching. We have shown that it is possible to extend the WARREN Abstract Machine to feature unification. We have implemented this extended Prolog machine prototypically, however there will be a lot of improvements of runtime behavior.

Feature Prolog has applications in computational linguistic and knowledge representation. E.g. with Feature Prolog we can represent Functional Unifications Grammars (FUG) /Per 87/ very efficient and natural. The burden of representation falls in Feature Prolog much more heavily on descriptions than on rules.

Acknowledgements

Hans-Jürgen Bürckert and Gerd Smolka suggested important corrections and improvements of presentation. I also thank Carla Decker, Michael Dreiucker and Michael Tepp for disposing me the implementation of the WARREN Abstract Machine with the sort extensions. Special thanks to Michael Tepp for many discussions that helped build my understanding of the WARREN Abstract Machine. The remaining errors and obscurities are, of course, my own.

Appendix

In the following we give the compiled code of the entrance example in the Feature Prolog version. The generated code is modified in that sense that all internal symbols are converted into the user defined symbols.

The Warren Code of example 1b:

```
grandfather/2        try_me_else grandfather/2-1
                     get_fvariable_void 0 person 1
                     unify_last_feature_x_variable mother 10
                     get_fvariable_void 10 person 1
                     unify_last_feature_x_svariable father person 11
                     get_X_value 11 1
grandfather/2-1      trust_me_else
                     get_fvariable_void 0 person 1
                     unify_last_feature_x_variable father 10
                     get_fvariable_void 10 person 1
                     unify_last_feature_x_svariable father person 11
                     get_X_value 11 1
                     proceed
sign_of_zodiac/2     try_me_else sign_of_zodiac/2-1
                     get_fvariable_void 0 person 1
                     unify_last_feature_x_var month_of_birth 10
                     s_get_constant january 10
                     s_get_constant capricorn 1
                     proceed
sign_of_zodiac/2-1   try_me_else sign_of_zodiac/2-2
                     get_fvariable_void 0 person 1
                     unify_last_feature_x_var month_of_birth 10
                     s_get_constant february 10
                     s_get_constant aquarius 1
                     proceed
sign_of_zodiac/2-... try_me_else sign_of_zodiac/2-...


QUERY0               allocate 3
                     put_fconstant peter 0
                     s_put_Y_variable 0 1 person
                     put_Y_local_value 0
                     call grandfather/2 3
                     put_unsafe_value 0 0
                     s_put_constant pisces 1
                     deallocate
                     execute sign_of_zodiac/2
```

The next compiled code is from the source code given in Example2. It's the original code of the implemented machine.

```
;---------------THE IDENT-TABLE--------------;
(73 (73 . |course_list|) (48 . |no_curriculum|) (47 . |teacher|) (72 . |jobtitle|) (46 . |professor|) (45
 . |comp3|) (44 . |comp2|) (43 . |comp1|) (42 . |ma3|) (41 . |ma2|) (40 . |ma1|) (39 . |pe3|) (38 . |pe2|
) (37 . |pe1|) (36 . |cs3|) (35 . |cs2|) (34 . |cs1|) (33 . |chem2|) (32 . |chem1|) (31 . |bio3|) (71 . |
curriculum|) (30 . |bio2|) (29 . |abdula|) (28 . |mary|) (70 . |first_name|) (27 . |peter|) (26 . |high_i
ncome2|) (25 . |high_income1|) (24 . |middle_income3|) (23 . |middle_income2|) (22 . |middle_income1|) (2
1 . |low_income3|) (20 . |low_income2|) (19 . |low_income1|) (18 . |f2|) (17 . |f1|) (16 . |e2|) (15 . |e
1|) (14 . |w2|) (13 . |w1|) (12 . |s2|) (11 . |s1|) (10 . |no_eg_member|) (9 . |local|) (69 . |status|) (
8 . |eg_member|) (68 . |physical_education|) (67 . |computer_science|) (66 . |chemestry|) (65 . |subject|
) (64 . |biology|) (63 . |old|) (62 . |middle|) (61 . |age|) (60 . |young|) (59 . |high_income|) (58 . |m
iddle_income|) (57 . |income|) (56 . |low_income|) (55 . |workstudy|) (54 . |faculty|) (53 . |staff|) (5
2 . |employee|) (51 . |person|) (50 . |student|) (49 . |any|))


;---------------THE SORT-TABLE---------------;
((48 . 73) (47 . 72) (46 . 72) (45 . 71) (44 . 71) (43 . 71) (42 . 71) (41 . 71) (40 . 71) (39 . 71) (38
. 71) (37 . 71) (36 . 71) (35 . 71) (34 . 71) (33 . 71) (32 . 71) (31 . 71) (30 . 71) (29 . 70) (28 . 70)
 (27 . 70) (26 . 59) (26 . 59) (25 . 59) (24 . 58) (23 . 58) (22 . 58) (21 . 56) (20 . 56) (19 . 56) (18
. 54) (17 . 54) (16 . 53) (15 . 53) (14 . 55) (13 . 55) (12 . 50) (11 . 50) (10 . 69) (9 . 69) (8 . 69) (
68 . 65) (67 . 65) (66 . 65) (64 . 65) (63 . 61) (62 . 61) (60 . 61) (59 . 57) (58 . 57) (56 . 57) (55 .
53) (55 . 50) (54 . 52) (53 . 52) (52 . 51) (50 . 51) (51 . 49) (57 . 49) (61 . 49) (65 . 49) (69 . 49) (
70 . 49) (71 . 49) (72 . 49) (73 . 49))


;---------------THE FEATURE-LIST-------------;
((|salary| . 8) (|position| . 7) (|major| . 6) (|father| . 5) (|mother| . 4) (|first_id| . 3) (|social_st
atus| . 2) (|period_of_life| . 1))


;----------THE CONSTANT PROPERTY LIST--------;
(49 9 (48) (47) (46) (45) (44) (43) (42) (41) (40) (39) (38) (37) (36) (35) (34) (33) (32) (31) (30) (29)
 (28) (27) (26) (26) (25) (24) (23) (22) (21) (20) (19) (18 (1 . 61) (2 . 69) (3 . 70) (4 . 51) (5 . 51)
(7 . 72) (8 . 57)) (17 (1 . 61) (2 . 69) (3 . 70) (4 . 51) (5 . 51) (7 . 72) (8 . 57)) (16 (1 . 61) (2 .
69) (3 . 70) (4 . 51) (5 . 51) (7 . 72) (8 . 57)) (15 (1 . 61) (2 . 69) (3 . 70) (4 . 51) (5 . 51) (7 . 7
2) (8 . 57)) (14 (1 . 62) (2 . 10) (3 . 29) (4 . 51) (5 . 51) (6 . 67) (7 . 72) (8 . 57)) (13 (1 . 62) (2
 . 9) (3 . 28) (4 . 51) (5 . 51) (6 . 68) (7 . 72) (8 . 57)) (12 (1 . 62) (2 . 8) (3 . 27) (4 . 51) (5 .
51) (6 . 66)) (11 (1 . 62) (2 . 9) (3 . 27) (4 . 51) (5 . 51) (6 . 64)) (10) (9) (8))


;-----THE (SORT - NR. OF PROPERTIES) LIST----;
((13 . 8) (14 . 8) (17 . 7) (18 . 7) (55 . 8) (15 . 7) (16 . 7) (53 . 7) (54 . 7) (11 . 6) (12 . 6) (50 .
 6) (52 . 7) (51 . 5))
```

```
|foreign/1|
(WPM-GET_FVAR_VOID 0 51 1)
(WPM-UNIFY_LAST_FEATURE_X_VAR 2 10)
(WPM-S_GET_CONSTANT 10 10)
(WPM-PROCEED)
|few_courses/1|
(WPM-TRY_ME_ELSE |few_courses/1-1| 1)
(WPM-S_GET_CONSTANT 48 0)
(WPM-PROCEED)
|few_courses/1-1|
(WPM-RETRY_ME_ELSE |few_courses/1-2|)
(WPM-S_GET_STRUCTURE (QUOTE (|cons| 2)) 0 73)
(WPM-S_UNIFY_X_VARIABLE 10 71)
(WPM-S_UNIFY_CONSTANT 48)
(WPM-PROCEED)
|few_courses/1-2|
(WPM-RETRY_ME_ELSE |few_courses/1-3|)
(WPM-S_GET_STRUCTURE (QUOTE (|cons| 2)) 0 73)
(WPM-S_UNIFY_X_VARIABLE 10 71)
(WPM-UNIFY_X_VARIABLE 11)
(WPM-S_GET_STRUCTURE (QUOTE (|cons| 2)) 11 73)
(WPM-S_UNIFY_X_VARIABLE 12 71)
```

```
(WPM-S_UNIFY_CONSTANT 48)
(WPM-PROCEED)
|few_courses/1-3|
(WPM-TRUST_ME_ELSE)
(WPM-S_GET_STRUCTURE (QUOTE (|cons| 2)) 0 73)
(WPM-S_UNIFY_X_VARIABLE 10 71)
(WPM-UNIFY_X_VARIABLE 11)
(WPM-S_GET_STRUCTURE (QUOTE (|cons| 2)) 11 73)
(WPM-S_UNIFY_X_VARIABLE 12 71)
(WPM-UNIFY_X_VARIABLE 13)
(WPM-S_GET_STRUCTURE (QUOTE (|cons| 2)) 13 73)
(WPM-S_UNIFY_X_VARIABLE 14 71)
(WPM-S_UNIFY_CONSTANT 48)
(WPM-PROCEED)
|takes/2|
(WPM-TRY_ME_ELSE |takes/2-1| 2)
(WPM-GET_FCONST 11 0)
(WPM-S_GET_STRUCTURE (QUOTE (|cons| 2)) 1 73)
(WPM-S_UNIFY_CONSTANT 40)
(WPM-UNIFY_X_VARIABLE 10)
(WPM-S_GET_STRUCTURE (QUOTE (|cons| 2)) 10 73)
(WPM-S_UNIFY_CONSTANT 33)
(WPM-UNIFY_X_VARIABLE 11)
(WPM-S_GET_STRUCTURE (QUOTE (|cons| 2)) 11 73)
(WPM-S_UNIFY_CONSTANT 30)
(WPM-UNIFY_X_VARIABLE 12)
(WPM-S_GET_STRUCTURE (QUOTE (|cons| 2)) 12 73)
(WPM-S_UNIFY_CONSTANT 31)
(WPM-S_UNIFY_CONSTANT 48)
(WPM-PROCEED)
|takes/2-1|
(WPM-RETRY_ME_ELSE |takes/2-2|)
(WPM-GET_FCONST 12 0)
(WPM-S_GET_STRUCTURE (QUOTE (|cons| 2)) 1 73)
(WPM-S_UNIFY_CONSTANT 32)
(WPM-UNIFY_X_VARIABLE 10)
(WPM-S_GET_STRUCTURE (QUOTE (|cons| 2)) 10 73)
(WPM-S_UNIFY_CONSTANT 33)
(WPM-UNIFY_X_VARIABLE 11)
(WPM-S_GET_STRUCTURE (QUOTE (|cons| 2)) 11 73)
(WPM-S_UNIFY_CONSTANT 30)
(WPM-UNIFY_X_VARIABLE 12)
(WPM-S_GET_STRUCTURE (QUOTE (|cons| 2)) 12 73)
(WPM-S_UNIFY_CONSTANT 42)
(WPM-S_UNIFY_CONSTANT 48)
(WPM-PROCEED)
|takes/2-2|
(WPM-RETRY_ME_ELSE |takes/2-3|)
(WPM-GET_FCONST 13 0)
(WPM-S_GET_STRUCTURE (QUOTE (|cons| 2)) 1 73)
(WPM-S_UNIFY_CONSTANT 40)
(WPM-UNIFY_X_VARIABLE 10)
(WPM-S_GET_STRUCTURE (QUOTE (|cons| 2)) 10 73)
(WPM-S_UNIFY_CONSTANT 41)
(WPM-UNIFY_X_VARIABLE 11)
(WPM-S_GET_STRUCTURE (QUOTE (|cons| 2)) 11 73)
(WPM-S_UNIFY_CONSTANT 42)
(WPM-UNIFY_X_VARIABLE 12)
(WPM-S_GET_STRUCTURE (QUOTE (|cons| 2)) 12 73)
(WPM-S_UNIFY_CONSTANT 38)
(WPM-S_UNIFY_CONSTANT 48)
(WPM-PROCEED)
|takes/2-3|
(WPM-TRUST_ME_ELSE)
(WPM-GET_FCONST 14 0)
(WPM-S_GET_STRUCTURE (QUOTE (|cons| 2)) 1 73)
(WPM-S_UNIFY_CONSTANT 40)
(WPM-UNIFY_X_VARIABLE 10)
(WPM-S_GET_STRUCTURE (QUOTE (|cons| 2)) 10 73)
(WPM-S_UNIFY_CONSTANT 44)
(WPM-S_UNIFY_CONSTANT 48)
(WPM-PROCEED)
|part_time/1|
(WPM-ALLOCATE 3)
(WPM-S_GET_X_VARIABLE 10 0 50)
(WPM-PUT_X_VALUE 10 0)
```

```
(WPM-S_PUT_VARIABLE_VOID 1 73)
(WPM-CALL |takes/2| 3)
(WPM-S_PUT_VARIABLE_VOID 0 73)
(WPM-DEALLOCATE)
(WPM-EXECUTE |few_courses/1|)
|query/1|
(WPM-ALLOCATE 2)
(WPM-S_GET_X_VARIABLE 10 0 70)
(WPM-PUT_Y_FVAR 0 0 55 2)
(WPM-UNIFY_FEATURE_X_VALUE 3 10)
(WPM-UNIFY_LAST_FEATURE_SVAR_VOID 8 56)
(WPM-CALL |foreign/1| 2)
(WPM-PUT_UNSAFE_VALUE 0 0)
(WPM-DEALLOCATE)
(WPM-EXECUTE |part_time/1|)
QUERY0
(WPM-VAR-MEM "X")
(WPM-S_PUT_VARIABLE_VOID 0 70)
(WPM-EXECUTE |query/1|)
"end"
((|query/1| . 1) (|part_time/1| . 1) (|takes/2| . 2) (|few_courses/1| . 1) (|foreign/1| . 1))
"end"
```

References

/Ait-Kaci 86/
Ait-Kaci H., Nasr R.
"LOGIN, A Logic Programming Language With Built-in Inheritance"
The Journal of Logic Programming (1986)3, 185-215

/AlFr 82/
Allen, J.F., and Frish A.M.
"What's in a Semantic Network",
In Proceedings of the 20th Annual ACL Meeting.
Association for Computational Linguistics, 1982

/Be 85,1/
Beer J.
"Comments on Compiling PROLOG Programs Using WARREN's Abstract PROLOG Instruction Set"
GMD-FIRST, Berlin 1985

/Be 85,2/
Beer J.
"An Extended PROLOG Instruction set for the PIPE-Report",
GMD-FIRST, Berlin 1985

/BrFi 83/
Brachman, R.E., Fikes, R.E, and Lavesque, H.J.
"KRYPTON: A Functional Approach to Knowledge Representation",
FLAIR Technical Report No. 16, Fairchild Lab. for Artificial Intelligence Research, Fairchild Research Center. Palo Alto, May 1983.

/Bue 85/
Bürckert H.J.
"Extending the WARREN Abstract Machine to Many-Sorted Prolog",
Seki-85-VII-KL, University of Kaiserslautern 1985

/DeKo 79/
Deliyanni, A., and Kowalski, R. A.
"Logic and Semantic Networks",
Communications of the ACM, 22(3):184-92. 1979.

/GLLO 84/
Gabriel John, Lindholm Tim, Lusk E.L., Overbeck R.A.
"A Tutorial on the Warren Abstract Machine for Computational Logic",
Mathematics and Computer Science Division
Argonne National Laboratories, 1984/85

/Hu 82/
"MPROLOG-Manual",
Institute for Coordination of Computer Techniques Budapest 1982

/Hub 85/
Huber M.
"L-Maschine: Maschinenmodell mit Sorten."
Arbeitsbericht, Universität Karlsruhe, 1985

/Per 87/
Pereira Fernando C.N.
"Grammars and Logics of Partial Information",
in Proceedings 4th. ICCP 1987, pp: 989-1012

/PIPE 84+85/
Parallel Inferencing PROLOG Environment
"Workshops and Reports of the PIPE-Project",
Berlin-Kaiserslautern-Paderborn 1984/85

/Ro 65/
Robinson J.A.
"Computitional Logic: The Unification Computation",
Machine Intelligence 6,1965

/Sch 85/
Schmidt-Schauß M.
"A Many-sorted Calculus with Polymorphic Functions Based on Resolution and Paramodulation"
Proceeding 9th IJCAI (1985) W. Kaufmann

/Si 84/
Siekmann J.
"Universal Unification",
Proceeding Conference of Automated Deduction, Los Angeles 1984

/SkiMi 79/
McSkimin, J.R., and Minker, J.
"A Predicate Calculus Based Semantic Network for Question-Answering Systems",
Associative Networks-The Representation and Use of Knowledge by Computers, Findler,N (ED.). Academic Press, New York, 1979.

/Smolka Ait-Kaci 87/
Smolka G. and Ait-Kaci H.
"Inheritance Hierarchies: Semantics and Unification",
MCC Technical Report AI-057-87, 1987

/Var 87/
Varsek I.
"Taxonomical Reasoning in Logic Programming"
in Proceeding of '3.Östereichische Artificial Intelligence-Tagung', 1987

/Wal 87/
Walter C.
"Many-Sorted Calculus based on Resolution and Paramodulation"
Research Notes in ARTIFICIAL INTELLIGENCE (1987) Pitman Publ., Ltd., London, and Morgan Kaufmann Publ., Inc., Los Altos (fort-hcoming)

/War 77/
Warren D.H.D.
"Compiling Predicate Logic Programms",
D.A.I. Research Report, University of Edinburgh, 1977

/War 83/
Warren D.H.D
"An Abstract PROLOG Instruction Set",
SRI Technical Report 309, 1983