Report of the Logic Programming Language PROTOS-L (revised version)

Christoph Beierle[1], **Stefan Böttcher**[2], **Gregor Meyer**[1]

[1] FernUniversität Hagen, Fachbereich Informatik
D-58084 Hagen, Germany
e-mail: {*Christoph.Beierle, Gregor.Meyer*}*@fernuni-hagen.de*

[2] Fachhochschule Ulm

June 19, 1991; revised July 1994

Abstract: PROTOS-L is a language based on logic programming that integrates a variety of concepts for programming large knowledge based systems like a powerful type concept, a module concept, high-level access to external relational databases, and finite domain constraints. The type concept covers user defined sorts, subsort relationships supporting multiple inheritance and parameterized sorts in the form of polymorphism. In addition to relations, also user-defined functions are available. The module concept is similar to that of Modula-2 and allows to hide implementation details from the user of a module. Database access and modification is fully embedded in the programming language PROTOS-L and can be programmed transparent to the user of a program or a program part. Besides simple links to external relations, non-recursive as well as recursive function free deduction rules can be defined. PROTOS-L also provides an easy way to work with windows via an object-oriented interface to the OSF/Motif system. These features as well as file handling, term manipulation etc. are embedded type-safe into PROTOS-L.

The implementation of PROTOS-L is based on the PROTOS Abstract Machine, an extension of the Warren Abstract Machine (WAM). In particular, it supports the required polymorphic order-sorted unification. It communicates with a database inference engine realizing a deductive database component, and also with a window manager realizing the interface to OSF/Motif.

This report describes the language PROTOS-L and its usage in the PROTOS-L prototype system in its current state of development. It is a working document in the sense that parts of it will be revised or extended, and that PROTOS-L is still being developed further. Thus, all comments and suggestions are greatly appreciated.

[1]The research reported here was started while the authors were with the Scientific Center of IBM Germany. This is a revised version of the original language report [Beierle *et al.*, 1991a] with revisions made by the first and the third author under a research contract with the Institute for Logics and Linguistics, Scientific Center, IBM Deutschland Informationssysteme GmbH.

This work was partially funded by the German Federal Ministry for Research and Technology (BMFT) in the framework of the WIPSRO Project under Grant 01 IW 206. The responsibility for the contents of this study lies with the authors.

Contents

0 Preface to the revised version

This is a revised version of the original report [Beierle *et al.*, 1991a]. The revisions were made by the first and the third author and concern the following aspects.

0.1 Finite domain constraints

A finite domain constraint solver (in the spirit of constraint logic programming) has been integrated into PROTOS-L. The constraint solver itself was developed at the IBM Scientific Center in Paris [Beringer and De Backer, 1994]. It is accessible via various built-in predicates and functions.

0.2 Interface to C

An interface to the programming language C is now available. It allows calls to C within PROTOS-L code as well as calling PROTOS-L from C.

0.3 Term databases

Term databases support the storing of terms in local databases. Modifications on term databases are not undone upon backtracking. Like all other built-in features such as file input and output, also the access to the term databases is completely type safe.

However, these PROTOS-L extensions are not described here but in the PROTOS-L User's Manual [Meyer and Beierle, 1994]. Furthermore, the following modifications where made in this revised version of the language report:

- Sections 8, 9, and 10 now contain only brief summaries of the file handling, the OSF/Motif Interface and the available built-ins. A detailed description of the current version of these features is given in [Meyer and Beierle, 1994].
- Corrections of the text and of the syntax (Section 11) were made, and various language restrictions that were still valid in [Beierle *et al.*, 1991a] have been removed. However, apart from these changes no attempt was made to update e.g. the comparison to the related work which has appeared in the meantime. Thus, also the references are still the same as in the previous version, apart from adding the references mentioned above and [Beierle, 1992], [Beierle and Börger, 1992], [Meyer *et al.*, 1994], [Beierle and Meyer, 1994] which provide some more information on different aspects of the PROTOS-L system.

Hagen, July 1994 C. Beierle and G. Meyer

1 Introduction

1.1 Overview

Prolog, the most prominent representative of the family of logic programming languages, lacks many of the features that play a predominant role in many other programming languages, e.g. a type concept or a module concept. Furthermore, as application systems become more and more complex, involving very large amounts of data and the embedding in an open system architecture with possibly many other software systems, the need for the integration of powerful and sophisticated interfaces to e.g. database management systems increases more and more.

PROTOS-L is an outcome of a research effort within the EUREKA Project PROTOS (Logic Programming Tools for Building Expert Systems, EU56) aimed at overcoming some of the shortcomings of Prolog. It provides various advanced features together with a state-of-the-art implementation approach based on an abstract machine.

PROTOS-L is a logic programming language based on typed Horn clause logic. The type concept which has been derived from TEL ([Smolka, 1988b], see the following subsection) allows for subtypes as well as for parametric polymorphism. Typing leads to better structured programs since they allow to make the data structures of a program explicit. They are exploited at compile time for static consistency checks, so that many programming errors can be detected early in the program development. The type information is also present at runtime through typed unification: Free variables can be constrained to subtypes without binding them to a particular value, and variables and terms can be tested for subtype membership. Moreover, the types can be compared to each other w.r.t. the type hierarchy. PROTOS-L also allows the definition of functions which are defined by a set of conditional equations. The mode discipline of PROTOS-L supports the checking of dataflow at compile time.

The purpose of the module concept of PROTOS-L is twofold: First, it provides a means for the structured development of large programs by supporting separate compilation of module interfaces and bodies. Furthermore, it provides a powerful means for the definition of abstract data types: At the level of interfaces the realization of types and operations on this type can be hidden.

In this way, modules also provide both a structured and transparent database access. Through a special kind of module bodies called database bodies an external relational database can be accessed, where this access is transparent at the level of the interface. Moreover, the inference rules in database bodies are interpreted by a deductive database component, combining advantages of relational databases, like efficient set-oriented evaluation, with advantages of the logic programming paradigm, like high-level programming and recursion.

Database access is completely type safe, as is file handling, array manipulation, etc.

Another highlight of PROTOS-L is the integration of an object-oriented interface to OSF/Motif. Based on this object oriented interface, high-level end-user interfaces can

be developed within PROTOS-L.

The implementation of PROTOS-L is based on the PROTOS inference engine (PIE) which consists of the PROTOS abstract machine PAM – an extension of the WAM ([Warren, 1983]), the database interpreter DBI and the PROTOS window manager PWM.

The PAM ([Semle, 1989], [Beierle *et al.*, 1991b]):

- realizes the necessary polymorphic order-sorted unification required by the type system,
- provides a set of type safe built-ins for file handling, array manipulation, string operations,
- offers a debug modus providing detailed information on its current machine state, etc., and
- communicates with both the DBI and the PWM.

The database inference engine DBI ([Meyer, 1989])

- controls the relational DBMS according to read access to relations, updates and transactions, and
- evaluates deduction rules which may be recursive; it is complete and terminating for Datalog (function free Horn clauses).

The PROTOS window manager PWM ([Jasper, 1991], [Schenk, 1991])

- provides a simple but flexible, object-oriented interface to OSF/Motif (AIX-Windows)

The PIE has been developed in C and runs under AIX 2.2.1 on IBM RT/PC 6150 coupled with SQL/RT, and under AIX 3.1 on IBM RS/6000 coupled with ORACLE and OSF/Motif.

The compiler for PROTOS-L supports the separate compilation and loading of modules, does the static type checking and type inference in order to detect type inconsistencies early in the program development, and produces machine code for the PIE. The compiler is currently implemented in the TEL system ([Smolka, 1988b], [Nutt and Smolka, 1993]) which runs on Quintus Prolog.

1.2 Related work

PROTOS-L integrates concepts developed in two different research streams: the enrichment of logic programming with types, and coupling logic programming with databases.

Comparing Prolog to other programming languages that are used for large systems not only the complete lacking of types in Prolog becomes apparent. Also major software

engineering principles like modularization or data abstraction are not directly supported in Prolog. Especially as applications grow larger and more complex such software engineering principles become more important. The basis for many of these principles can be given by a powerful type concept. It allows to detect many common programming errors at compile time which otherwise may be difficult to locate. Additionally, in AI applications like theorem proving it has been shown that the introduction of types with subtypes may drastically reduce the search space of a problem (e.g. [Walther, 1985]). On the other hand, types should not burden the programmer or knowledge engineer with requiring a too narrow and strict discipline. Therefore, the type concept of PROTOS-L covers user defined sorts, subsort relationships supporting multiple inheritance and parameterized sorts in the form of parametric polymorphism.

The design of PROTOS-L has been influenced by various other languages. In the starting point, there was a close connection to the development of the knowledge representation language L-LILOG ([Beierle *et al.*, 1989], [Pletat and v. Luck, 1990]) which, like PROTOS-L, is based on order-sorted first order predicate logic. Whereas L-LILOG supports full first order predicate logic and is much more liberal with respect to its type system e.g. by allowing sort expressions involving features and roles, PROTOS-L sticks to Horn clauses and sort constants, but allows additionally parametric sorts.

There are various approaches to enrich in particular logic programming with types (see e.g. [DeGroot and Lindstrom, 1986]). An integration of parametric polymorphism as in ML [Harper *et al.*, 1986] with Prolog was suggested in [Mycroft and O'Keefe, 1984]. Order-sorted type concepts were developed by Goguen et al. and are present in OBJ [Futatsugi *et al.*, 1985] and Eqlog [Goguen and Meseguer, 1986]. [Huber and Varsek, 1987] extends Prolog by order-sorted unification [Walther, 1988]. Polymorphism and order-sortedness are provided in the approach of [Dietrich and Hagl, 1988] by using modes to restrict the input/output data flow in certain cases. No such mode restrictions are required in the work of [Smolka, 1988a], [Smolka, 1989] where parametric polymorphism is combined with order-sortedness; this approach is the basis for the programming language TEL [Smolka, 1988b].

The type system of PROTOS-L has been derived from TEL which offers a slightly extended notion of subtypes and polymorphism: As opposed to TEL, polymorphic types in PROTOS-L may not have any explicitly defined subtypes ([Beierle, 1990]).

The availability of the TEL system also influenced the design of PROTOS-L: In order to be able to reuse TEL's compile time type, mode and module checking facilities, the current syntax of PROTOS-L for types, modules, relations, and functions is oriented towards TEL. Thus, the front-end of the PROTOS-L compiler could be derived from the TEL compiler's front-end by appropriate extensions and modifications. However, currently the TEL system does not offer a correct implementation of the required typed unification: The current implementation of TEL is not correct since it maps the runtime unification onto Quintus Prolog's ordinary term unification ([Smolka, 1988b], [Nutt and Smolka, 1993]). In contrast, the PROTOS-L compiler generates code for the PAM that provides a correct implementation of polymorphic order-sorted unification as required by PROTOS-L's type system.

No order-sortedness but another form of polymorphism is suggested in [Hanus, 1988], [Hanus, 1989]. The logic programming language Gödel ([Hill and Lloyd, 1992]) offers many-sorted types with parametric polymorphism; its main emphasis is to provide a declarative meaning for meta-logical facilities. For more complete surveys on type concepts for logic programming see e.g. [Hanus, 1988] or [Smolka, 1989].

The Abstract Machine of PROTOS-L is coupled to an external relational database by a separate component called DataBase Interpreter (DBI). A variety of coupling alternatives between logic programming and databases is discussed in [Vassiliou, 1986]. The DBI realizes a tight coupling in order to make intelligent use of the selection conditions at run time and to take advantage of the full power of the DB query processing facilities.

Some Prolog systems, e.g. IBM Prolog [IBM, 1989], provide an ad hoc means to access an external database by special built-in predicates which evaluate a query formulated in the DB query language like SQL or QUEL. In other systems the DB access is formulated transparently by deduction rules, often restricted to 'Datalog'. While only one level views can be formulated in Quintus Prolog [Quintus, 1987] other research systems allow multilevel views and recursion, e.g. [Bocca, 1986]. In PROTOS-L embedded SQL as well as (recursive) Datalog rules are allowed. At the level of module interfaces the database access is fully transparent.

A distinguishing feature of deductive databases is the ability to cope with recursively defined predicates. There are many algorithms and ideas how to evaluate rules on a DBMS ranging from top down Prolog like evaluation over variations of the Magic Set approach [Bancilhon *et al.*, 1986] to pure bottom up methods [Bayer, 1985, Bancilhon, 1986]; [Bancilhon and Ramakrishnan, 1986] gives a good overview. The DBI in PROTOS-L uses a mixed top down and bottom up strategy, combining ideas from [Vieille, 1988] and [Hulin, 1989]. For a more complete overview on combining logic programming systems with databases see [Ceri *et al.*, 1990].

Acknowledgements: PROTOS-L was developed within the EUREKA Project PROTOS (EU 56) at IBM in Stuttgart. Many people contributed directly or indirectly to its design and implementation. As research staff members and guest scientists Udo Pletat and Kurt Rothermel contributed to a first design, Gert Smolka brought in his experience with the TEL system and polymorphic order-sorted unification, and Heinrich Jasper developed an approach for the integration of the OSF/Motif system. A great deal of design and development work was done in the framework of a number of project and Diploma theses that were carried out in the framework of the PROTOS project. Bernd Müller completed the first order-sorted version of the PROTOS Abstract Machine. Heiner Semle implemented the polymorphic version of the PAM and a first version of a compiler for PROTOS-L, and made various additional contributions to the development of the PROTOS system. Martin Zeller realized the module concept in the code generator of the compiler. Irmgard Hauner implemented a first database coupling for PROTOS-L. In his PhD thesis, Christos Garidis made a study how a clustering concept of Prolog code can help to reduce dynamic loading at run time, and Jürgen Zink implemented an evaluation system for it. Michael Schenk implemented the interface to the OSF/Motif system. Holger Wittman and Gerhard Urban developed some advanced applications in PROTOS-L, and Ralph Scheubrein worked on the coupling with the non-standard

database system LILOG-DB. The majority of our Diploma students also worked as summer students at various times in the project, as well as, among others, Ralf Hauser, Frank Hunziker, Andreas Molitor, Peter Sanders, Ralf Scheidhauer, Hinrich Schütze, Markus Walther, and Oliver Wauschkuhn. The work of all of them greatly contributed to the success of the PROTOS project. Thanks also go to U. Geske and H.-J. Goltz who made some detailed and helpful comments on an earlier version of this paper.

2 Types

There are two different forms of type definitions in PROTOS-L, monomorphic and polymorphic ones.

type_definition $\longrightarrow$
mono_type_definition
| *poly_type_definition*

Each type definition consists of a left-hand side that contains in particular the name of the type and a right-hand side that defines (directly or indirectly) the elements of that type.

A note on notation: We will present parts of the formal syntax definition of PROTOS-L in the various sections (a complete syntax definition will be given in Section 11) for which we use *slanted* type face as in the syntax rule above. PROTOS-L program text is printed in ordinary `type writer` face. Like in Prolog, constants, predicate names, etc. start with a lower case letter, whereas variables start with an upper case letter.

2.1 Monomorphic Types

A simple monomorphic type definition is

```
car := { ford, opel, mercedes }.
```

which defines the *type constant* `car`. The type `car` has exactly the three elements denoted by the value constants `ford`, `opel`, and `mercedes`. These constants are also called the (value) constructors for the type `car`, and `car` is the *least type* of them.

Given another type definition

```
airplane := { boing747, dc10, airbus }.
```

one can also define the union of types like

```
vehicle := car ++ airplane.
```

The last definition states that `car` and `airplane` are *subtypes* of the type `vehicle`. Thus, e.g. `opel` and `dc10` are both also of type `vehicle` but their least type is `car` and `airplane`, respectively.

So far, the constructors of a type have all been constants. The type definition

```
stack := { empty,
           push:  car x stack }.
```

introduces a constant constructor

```
empty:  → stack
```

and a binary constructor

```
push:  car x stack → stack
```

taking two arguments of type `car` and `stack`, respectively, and yielding a term of type `stack`. Thus,

```
empty
push(ford, empty)
push(opel, push(ford, empty))
```

are all terms of type `stack`. However, given the definitions above the term

```
push(dc10, push(ford, empty))                                    (1)
```

would be ill-typed. But if the definition of stack were changed to

```
stack := { empty,
           push:  vehicle x stack }.
```

then (1) would be a well-typed term of type `stack`.

In a single type definition one can also have both subtypes and the definition of constructors, e.g.

```
vehicle := car ++ airplane ++
           { train:  train_type,
             bike:  bike_type }.
```

where e.g. `train_type` could be

```
train_type := { local, intercity, eurocity }.
```

Then

```
train(intercity)
opel
airbus
```

are all terms of type `vehicle`, their least type being `vehicle`, `car`, and `airplane`, respectively.

Note that every (ground) term and thus in particular every constant must have a unique least type. Thus, it is *not* allowed to define

```
car      := { flying_car, surface_plane, ford, opel, mercedes }.
airplane := { flying_car, surface_plane, boing747, dc10, airbus }.
```

since e.g. flying_car would not have a unique least type. On the other hand, it is possible to define the intersection of types since a type may have more than one supertype:

```
drive_and_fly := { flying_car, surface_plane }.
car           := drive_and_fly ++ { ford, opel, mercedes }.
airplane      := drive_and_fly ++ { boing747, dc10, airbus }.
```

In this case the least type of `flying_car` is `drive_and_fly` while the least type of `ford` is still `car`.

Another technical condition is that if two types have more than one common subtype then there must be a greatest common subtype, i.e. the sort hierarchy must form a semi-lattice (c.f. [Walther, 1988]). This condition ensures the completeness of the underlying deduction mechanism and can be satisfied easily by introducing a new type where necessary.

All declarations and definitions in PROTOS-L come in modules (see Section 5). Each (monomorphic) type may be used as a subtype within the same module. If a type is not used as a subtype within the module where it is introduced it is automatically assumed that this type is a maximal type that will not be used as a subtype. However, if one does want to use such a type `t` that is maximal in its own module as a subtype in another module then `t` has to be declared as a *non-maximal type* which is indicated by the key word `nmtype`.

For instance, the definition

```
nmtype vehicle := car ++ airplane.
```

introduces `vehicle` as a non-maximal type that may occur in a subtype position in any other module. The distinction between maximal types and non-maximal types is made not only for readability of the program, but also in order to support faster program execution since type information for maximal types can be neglected at runtime ([Smolka, 1989], [Beierle, 1990]). In particular, this means that if all types are maximal then no type information is necessary at run-time, a situation that corresponds to ordinary Prolog.

The subtype relationship may not introduce any cycle, i.e. a type is never a proper subtype of itself.

In the formal syntax definition for monomorphic type definitions we use the following notation:

- A terminal form `'T'` means that the token `T` must appear physically.
- The symbol `'|'` separates alternatives.
- An optional form [F] means that the form F is optional.
- A list form {F} means that the form F appears either once or more than once separated by commas `','` .
- A star form (F)* denotes a possibly empty sequence of Fs.

mono_type_definition $\longrightarrow$
mono_type_dec_lhs `':='` *mono_type_def_rhs* `'.'`

mono_type_dec_lhs $\longrightarrow$
[`'nmtype'`] *type_identifier*

mono_type_def_rhs ⟶

(*subtype_specification* '++')*
subtype_specification '++'
subtype_specification
| (*subtype_specification* '++')*
'{' {*mono_constructor_def*} '}'

note that in the actual PROTOS-L system there may be a restriction on the maximal number of direct subtypes

subtype_specification ⟶

type_identifier

Examples for monomorphic constructor definitions were already given above in the alternative definitions for `stack`. However, in the general case the argument type of a monomorphic constructor may not only be a type identifier but also a ground instance of a *polymorphic* type (c.f. Section 2.2). For example,

```
stack := { empty,
           push:  list(car) x stack }.
```

introduces a two-argument constructor

```
push:  list(car) x stack → stack
```

where `list(car)` is a ground instance of the polymorphic type `list`. `list(car)` is called a (ground) *type term*. Other examples of such type terms are

```
list(vehicle)
list(airplane)
pair(car, list(vehicle))
```

We close the introduction of monomorphic types with the syntax definition of monomorphic type constructors:

mono_constructor_def ⟶

constr_designator ['**:**' *ground_domain*]

constr_designator ⟶

identifier

ground_domain ⟶

ground_type_term ['x' *ground_domain*]

2.2 Polymorphic Types

In a typed language the definition of parametric types plays an important role. For instance, one wants to have typed lists, i.e. lists over different ground types. In PROTOS-L

this is achieved by so-called *polymorphic* types. A polymorphic type has parameter variables that range over the set of types. In the previous subsection we gave three alternative definitions for a type `stack` where the types of the elements in the stack were `car`, `vehicle`, and `list(car)`, respectively.

A more economic and flexible way is to introduce a polymorphic type definition

```
stack(T) := { empty,
              push:  T x stack(T) }.
```

In this definition, the variable `T` is a type variable which stands for any type or type term. Syntactically, type variables start with a capital letter, just as ordinary variables. The constructor

```
push:  T x stack(T) → stack(T)
```

is a polymorphic (value) constructor that yields a term of type `stack(T)`.

All polymorphic types are monotonic with respect to the subtype relationship. If `car` is a subtype of `vehicle` then `stack(car)` is a subtype of `stack(vehicle)`. Here are some terms together with their least type:

a term	:	*its least type*
`push(ford, empty)`	:	`stack(car)`
`push(opel, push(ford, empty))`	:	`stack(car)`
`push(boing747, empty)`	:	`stack(airplane)`

In general, a term `push(t1, push(t2, empty))` is only a well-typed term if there is a type such that both `t1` and `t2` are of this type (which is not necessarily the least type of any of the two terms).

A built-in type in PROTOS-L is the polymorphic list type

```
list(T) := { nil,
             .  :  T x list(T) }.
```

where the list constructor '`.`' is treated as a right-associative infix operator. Thus, the list containing the two elements `opel` and `ford` is written as

```
opel.ford.nil                                          (2)
```

instead of `.(opel,.(ford,nil))`.

The type `list(car)` is a subtype of `list(vehicle)`, and `list(list(car))` is a subtype of `list(list(vehicle))` according to the monotonicity of polymorphic types. The least type of (2) is `list(car)`, but the least type of

```
opel.ford.airbus.nil
```

is `list(vehicle)`.

From a computational point of view it is desirable to be able to reduce the subtype relationship between instances of polymorphic types to the relationship between the type constants (i.e. the monomorphic types). Therefore, polymorphic types or instances thereof may not occur in a subtype position in any type definition - this is the reason why a subtype

specification in a monomorphic type definition is a type identifier but not a type term such as `list(car)` (c.f. the syntax rules given above). On the other hand, polymorphic type definitions do not contain an explicit subtype specification and therefore, all polymorphic types are automatically maximal. These observations make up the differences between monomorphic and polymorphic type definitions:

poly_type_definition $\longrightarrow$

poly_type_dec_lhs `':='` `'{'` {*poly_constructor_definition*} `'}'` `'.'`

every variable that occurs in the left-hand side must occur in the right-hand side and vice versa

poly_type_dec_lhs $\longrightarrow$

type_identifier `'('` {*type_variable*} `')'`

the occurring variables must be pairwise distinct

poly_constructor_definition $\longrightarrow$

constr_designator [`':'` *domain*]

domain $\longrightarrow$

type_term [`'x'` *domain*]

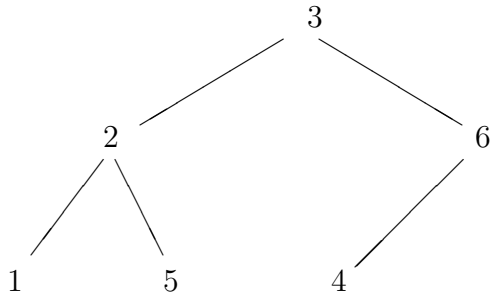
We close this subsection by giving an example definition for labelled trees. Each node in a tree is labelled with an element of a parameter type. A tree may consist just of a single node (which is a leaf), it may contain only a left or only a right subtree, and it may have both a left and a right subtree.

```
bin_tree(T) := { leaf:  T,
                 left:  bin_tree(T) x T,
                 right:  T x bin_tree(T),
                 both:  bin_tree(T) x T x bin_tree(T) }.
```

Now the term

```
both(both(leaf(1), 2, leaf(5)), 3, left(leaf(4), 6))
```

is a term of type `bin_tree(nat)` where `nat` is the built-in type of natural numbers {0, 1, 2, 3, ...}. It represents the binary tree



2.3 Built-in Types

There are several built-in types in PROTOS-L. Here we only give an overview (see also Section 10); a complete description of them is given in [Meyer and Beierle, 1994].

- Monomorphic types:

`nat`	natural numbers
`int`	integers
`zero`	contains only the number 0
`posint`	positive integers
`negint`	negative integers
`char`	characters
`string`	strings
`byte`	bytes used in file I/O (see also Section 8)
`pwm_object`	objects for Motif interface (see Section 9)
`pwm_attribute`	attibutes for Motif interface (see Section 9)
`pwm_goal`	call-back procedures for Motif interface (see Section 9)

- Polymorphic types:

`list(T)`	lists
`array(T)`	(one-dimensional) arrays
`instream(T)`	streams for file input (see also Section 8)
`outstream(T)`	streams for file output (see also Section 8)
`term_database(T)`	term databases

3 Relations

3.1 Relation declarations

A relation definition in Prolog consists of a sequence of clauses. The same is true for PROTOS-L but with the additional information that the types of the arguments of the relation are declared. For instance, the relation declaration

```
rel speed:  car x ?nat.
```

introduces a two-argument relation symbol `speed` taking a `car` term as first argument and a `nat` term as second argument. Additionally, data flow information is given: The first argument is declared as an input argument and the second argument is an output argument, indicated by the question mark '`?`'. At calling time the input arguments should not contain any variables, and after evaluation the arguments in an output position will be ground, i.e. they will not contain any variables. The compiler enforces this discipline, but by declaring a variable as *open* (see below in 3.2) it is possible to pass also a variable into an input position and to obtain a non-ground value in an output position. This mechanism yields the flexible input-output behaviour which is of course one of the distinctive advantages of logic programming.

An example of a polymorphic relation declaration is

```
rel append:  list(T) x list(T) x ?list(T).
```

which declares the arity and the argument types of the usual append predicate. The question mark indicates that the third argument of `append` is an output argument while the first two arguments are input arguments.

Here is the syntax of a relation declaration which is part of a complete relation definition:

relation_definition $\longrightarrow$
relation_declaration
relational_clause
(*relational_clause*)*

relation_declaration $\longrightarrow$
rel_class relation_designator '`:`' *io_domain* '`.`'
| '`proc`' *relation_designator* '`.`'
| '`tproc`' *relation_designator* '`.`'

rel_class $\longrightarrow$

'rel'
| 'drel'
| 'trel'
| 'tdrel'
| 'proc'
| 'tproc'
| 'urel'
| 'trans'

io_domain $\longrightarrow$

['?'] *type_term* ['x' *io_domain*]

relation_designator $\longrightarrow$

identifier

For the discussion of relation classes we refer to Section 3.3; here we will only deal with "ordinary" relations which are indicated by the relation class designator `rel`. In the following, we will sometimes use the terms relation and predicate as synonyms.

Before moving to the definition of predicates with clauses let us look at some queries and how they are checked for type consistency with respect to the declarations. In the query

```
PROTOS> append(1.2.3.nil, 4.5.nil, L).
```

both the first and the second argument are of type `list(nat)` and thus, the type of the variable `L` is derived to be `list(nat)` also. Likewise, in the query

```
PROTOS> append(ford.nil, opel.nil, L).
```

both the first and the second argument are of type `list(car)` and thus, the type of the variable `L` is `list(car)`. In the query

```
PROTOS> append(ford.opel.nil, airbus.nil, L).
```

the first argument is of type `list(car)` and the second argument is of type `list(airplane)`. The common supertype of both arguments is `list(vehicle)` which is the type that is derived for L. However, the query

```
PROTOS> append(ford.opel.nil, 4.5.nil, L).
```

would be rejected as being ill-typed since `car` and `nat` and thus also `list(car)` and `list(nat)` do not have a common supertype. The query

```
PROTOS> append(1.2.3.nil, 4, L).
```

is also ill-typed since the second argument is not even a list.

3.2 Relation definitions

A simple relation declaration together with its definition is

```
rel speed:  car x ?nat.
    speed(opel, 120).
    speed(ford, 140).
    speed(mercedes, 160).
```

Here, the three defining clauses for `speed` are simple facts without a conditional part. Since the output in both arguments will always be ground independent of the input arguments, the modes for `speed` could just as well be declared as

```
rel speed:  ?car x ?nat.
```

in this case.

Here is an example of a polymorphic relation definition:

```
rel append:  list(T) x list(T) x ?list(T).
    append(nil, L, L).
    append(H.T, L, H.TL) <-- append(T, L, TL).
```

Thus, as usual, appending the empty list `nil` to any other list `L` yields `L`, and appending a list with head `H` and tail `T` to a list `L` yields the list with head `H` and tail `TL` if appending `T` and `L` yields `TL`.

Note that the two clauses of `append` are exactly as in ordinary Prolog. However, in Prolog a goal like

```
append(nil, 3, 3).                                    (3)
```

would succeed although `append` is supposed to operate on lists only. In PROTOS-L the additional typing information is used by the type checker to correctly reject (3) as being ill-typed. Other type checking examples have already been given above.

Since `append` is defined as a polymorphic predicate it is applicable to all instances of list. Here are some queries and their results that are computed by the PROTOS-L system:

```
PROTOS> append(1.2.3.nil, 4.5.nil, L).
L = 1.2.3.4.5.nil

PROTOS> append(ford.nil, opel.nil, L).
L = ford.opel.nil

PROTOS> append(ford.opel.nil, airbus.nil, L).
L = ford.opel.airbus.nil
```

Note that the first two arguments of `append` are declared as input arguments. However, one of the specialities of Prolog is that input and output arguments can be freely mixed and interchanged. Correspondingly, in our situation with mode declarations one may like to put a variable or a non-ground term in an input position, and also to be able to have e.g. a non-ground term in an output position. In order to indicate these cases, one can

declare a variable as being an *open* variable. Such a declaration has the effect that the mode checker treats the variable as being a ground term. Thus, it is possible to pass a variable into an input position or to obtain a non-ground term in an output position.

The syntax for such an open variable declaration is

```
!X
```

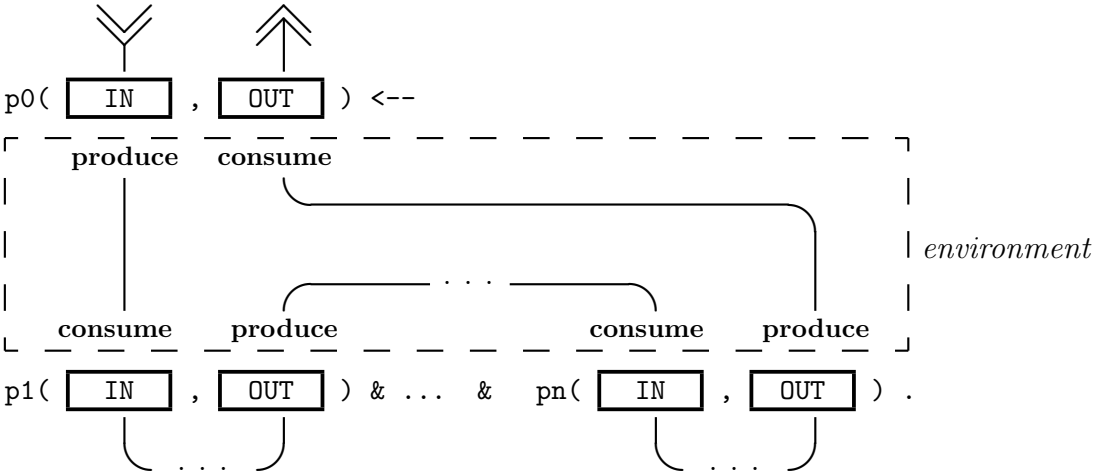
A PROTOS-L query involving open variables is

```
PROTOS> !L1 & !L2 & append(ford.L1, L2, ford.opel.mercedes.nil).
```

where `append` is not used for appending two given lists but where the list resulting from appending two lists is already given. Via backtracking PROTOS-L computes all possible solutions as demonstrated by the following dialog:

```
PROTOS> !L1 & !L2 & append(ford.L1, L2, ford.opel.mercedes.nil).
PAM started
L1 = nil
L2 = opel.mercedes.nil


MORE ANSWERS? (Y/N)? y
L1 = opel.nil
L2 = mercedes.nil


MORE ANSWERS? (Y/N)? y
L1 = opel.mercedes.nil
L2 = nil


MORE ANSWERS? (Y/N)? y
NO (MORE) ANSWERS

PROTOS>
```

In order to describe in more detail which variables in a PROTOS-L program have to be declared as open, we have to understand the principle of dataflow used in the mode checking algorithm: The head and each goal of a PROTOS-L clause divide the variables occurring in it into two disjunct groups: the input block containing all variables inside the input arguments of the corresponding predicate (those having no `'?'` in its declaration) and the output block containing all other variables. Note that it is possible for either block to be empty. Now look at the following diagram showing the dataflow graph of a single clause:



There are some entrypoints of the *environment* that are denoted by **produce** and some by **consume** saying that all variables contained in the corresponding input or output block are *produced* resp. *consumed* at this point of the dataflow graph. The mode checking algorithm requires that each variable that is *consumed* at a given position, needs to be *produced* at an earlier point in the dataflow graph. When this requirement is not fulfilled for any variable `X`, the programmer has to *produce* this variable before the critical consuming point by declaring it as open by `!X`.

Other examples of the use of open variables are given in the map colouring program in Section 5.

If there is a hierarchy of types it makes sense to test at runtime whether the value of a variable belongs to a certain subtype. For instance,

```
E:technician                                                    (4)
```

tests whether the value bound to `E` belongs to the type `technician`. Thus, (4) is called a `membership` condition. However, if `E` is still a free variable then (4) represents a very powerful mechanism of PROTOS-L, namely the possibility to constrain a variable to a subtype without binding it to a particular value of that type. For instance, suppose that the relation (for the complete example program see Appendix A)

```
rel can_repair :  technician x model.
    can_repair(T, M) <-- T:pc_technician & M:pc_model.
    can_repair(T, M) <-- T:mainframe_technician & M:mainframe_model.
```

is called with

```
can_repair(T, pc1)
```

where `T` is a free variable and `pc1` is of type `pc_model`. Then `T` is not bound to an element of type `pc_technician` but it is constrained to that type. Thus, the deduction process uses the more abstract level of set-denoting types rather than the level of individuals. This yields not only more compact intensional answers but it may also save a lot of expensive backtracking.

Similarly to the mode checking algorithm which sometimes requires a open variable declaration, the type checking algorithm will sometimes require that a variable `X` is restricted to a subtype by `X:s` if a following subgoal *consumes* `X` and expects `X` to be of type `s`. However, if the type checker has been able to automatically infer that `X` is of type `s` (or a subtype thereof), then no such declaration is necessary.

Further examples of the use of explicit subtype restrictions as well as the use of open variables are given Appendix A and B; in particular in the relations `can_repair` and `can_do_job` in A, `query` in B.1, `print_map_line` in B.2, `create_entries` in B.4, `all_countries` in B.5, or `complement` in B.6.

Apart from calling a relation, declaring a variable as open, and testing for membership there are currently the following other types of conditions that may occur in the condition part of a PROTOS-L relation definition:

- An equation of the form `t1 = t2` succeeds if `t1` and `t2` can be unified.
- A comparison of two arithmetic expressions like `X + Y < Z` requires that all variables occurring in it are bound to an integer value at runtime. The expressions are evaluated and compared w.r.t. the given relation symbol.
- A comparison of two `string` expressions where `string` is another built-in type.
- The "negation-as-failure" as known from Prolog which is denoted by `naf`.
- The "if-then-else" conditional: the condition `if a then b else c fi` is evaluated as follows: First, `a` is evaluated (only once - no backtracking will occur back into `a`). If `a` succeeds, `b` will be selected as the next subgoal, possibly with backtracking over `b`, but `c` will never be considered. Vice versa, if `a` fails, `b` will not be considered and `c` will be taken instead, again possibly with backtracking over `c`.

  Note that the operational semantics of the "if-then-else" conditional is defined as if it was a shorthand. I.e., a subgoal of the form

  ```
  if <a> then <b> else <c> fi
  ```

  is a shorthand for a subgoal defined by the two clauses

  ```
  ... <-- <a> & protos_cut & <b>.
  ... <-- <c>.
  ```

  where `protos_cut` corresponds to the the *cut* operator known from Prolog.
  (Warning: Thus be careful when using `protos_cut` explicitly in `<a>`, `<b>`, or `<c>` which will have a local effect.)
- Similar to the user-defined relation also a built-in predicate may be called (see Section 10).

Here is the formal syntax for a relational clause:

relational_clause $\longrightarrow$

relation_clause_head ['<--' *condition_part*] '.'

relation_clause_head ⟶
relation_designator [‘(’ {*var_term*} ‘)’]

relation_designator ⟶
identifier

condition_part ⟶
condition [‘&’ *condition_part*]

condition ⟶
relation_call
| *meta_predicate relation_call*
| *conditional*
| *variable* ‘islistof’ *var_term* ‘where’ *relation_call*
| *relation_call_parameter* ‘:’ *ground_type_term*
| *relation_call_parameter bool_op relation_call_parameter*
| ‘!’ *variable*

meta_predicate ⟶
‘naf’
| ‘insert’
| ‘delete’
| ‘dbassert’
| ‘dbdelete’

conditional ⟶
‘if’ *condition_part*
‘then’ *condition_part*
(‘elsif’ *condition_part* ‘then’ *condition_part*)*
[‘else’ *condition_part*]
‘fi’

relation_call ⟶
relation_designator [‘(’ {*relation_call_parameter*} ‘)’]

3.3 Relation classes

Relation declarations are usually introduced with the keyword `rel` as in

```
rel p :  ...
```

However, there are also additional types of relations.

The declaration

```
drel p :  ...
```

introduces a *deterministic* relation. Operationally, this means that no backtracking will occur on `p`. Once a clause of `p` has been completed successfully all other choices for `p` will be discarded. Thus, `p` will yield always at most one solution.

```
trel p :  ...
```

introduces a *total* relation. Operationally, this means that during runtime it is checked that every call of `p` eventually yields a success. If a call of `p` fails, a run-time error with an appropriate error message will be generated. (Note that also exhaustive backtracking over a non-deterministic total relation will eventually cause this run-time error to occur).

```
tdrel p :  ...
```

introduces a relation that is both total and deterministic.

```
urel p :  ...
```

introduces a relation that can be updated. This means that during runtime tuples may be added to or deleted from `p` dynamically by

```
insert p(...)
```

and

```
delete p(...)
```

respectively. (See Section 7 for more details on updatable relations).

```
trans p :  ...
```

introduces a *transaction* relation. The effect of inserting or deleting a tuple in an updatable relation is usually undone on backtracking. However, all operations within a transaction relation `p` will become permanent once `p` has been completed successfully. No backtracking will occur over `p` in this case; thus, a transaction relation is a special case of a deterministic relation.

For compatibility reasons with a previous version there are also relations which are classified as *proc*edures.

```
proc p :  ...
```

introduces such a relation and

```
tproc p :  ...
```

introduces a relation that is a total procedure. Any relation that contains a call to a `proc` or `tproc` relation must be itself a `proc` or `tproc` relation.

3.4 Some Built-in Relations

There are built-in relations dealing with types, arithmetic, file handling etc. In this subsection we only present some of the built-in relations that are not directly related to particular built-in types (like arithmetic or files); the latter are presented in the separate sections. All built-ins are described in the PROTOS-L User's Manual [Meyer and Beierle, 1994].

The built-in relation

```
rel var :  T.
```

succeeds if its argument is a free variable; otherwise it fails.

The built-in relation

```
rel nonvar :  T.
```

succeeds if its argument is not a free variable (regardless of any type restrictions for it); otherwise it fails.

The built-in relation

```
rel ground :  T.
```

succeeds if its argument is a ground term, i.e. it may not contain a free variable.

The built-in relation

```
rel nonground :  T.
```

succeeds if its argument is not a ground term; otherwise it fails.

The membership condition `X:t` test the actual type of `X` and may thereby change the type of `X` to a subtype. The next three built-in predicates also test the type of a variable (or more generally, of a term) but they never change the type of its argument.

The built-in relation

```
rel sort_eq:  T x ground_type_term.
```

checks whether the first argument has exactly the type given in the second argument. Thus,

```
!X & X:car & sort_eq(X, car).
```

succeeds, whereas

```
!X & X:car & sort_eq(X, vehicle).
```

fails.

The built-in relation

```
rel sort_le:  T x ground_type_term.
```

checks whether the first argument has a type which is a subtype of the second argument. Thus, both

```
!X & X:car & sort_le(X, vehicle).
!X & X:car & sort_le(X, car).
```

succeed, whereas

```
!X & X:vehicle & sort_le(X, car).
```

fails.

The built-in relation

```
rel sort_glb:  T x ground_type_term.
```

checks whether the type of its first argument and the type given in the second argument have a common subtype (i.e. a greatest lower bound exists). Thus, whenever `sort_le(X,t)` succeeds then also `sort_glb(X,t)` succeeds. In addition to that,

```
!X & X:car & sort_glb(X, boat).
```

would succeed in a type hierarchy where `amphibious_vehicle` is a subtype of both `car` and `boat`, whereas

```
!X & X:car & sort_glb(X, airplane).
```

fails if the two types have no common subtype.

The built-in relations

```
rel instantiate:  T x ?list(T).
rel instantiate1:  T x ?list(T).
```

take a variable in its first argument and yield the list of all possible instantiations. `instantiate` goes recursively also to the subsorts, whereas `instantiate1` yields a variable for each subsort instead. Thus,

```
!X & X:car & instantiate(X, L)
```

might yield

```
L = opel.ford.mercedes.amphi1.amphi2.nil
```

whereas

```
!X & X:car & instantiate1(X, L)
```

would give us

```
L = opel.ford.mercedes.A:amphibious_vehicle.nil
```

Additional built-in relations and functions are sketched in Section 10 and described in detail in [Meyer and Beierle, 1994].

4 Functions

4.1 Function declarations and definitions

A well-known disadvantage of purely relational notation is that functions with fixed input-output behaviour can not be written as such and that functional nesting is not available. Suppose we have a function

```
append:  list(T) x list(T) --> list(T).
```

that appends two lists and a predicate

```
rel p:  list(T) x nat.
```

Applying `p` to the result of appending two lists should be written as

```
...& p(append(L1, L2), N) & ...                    (1)
```

In relational style `append` would be a three-place relation and (1) would be written as

```
...& append(L1, L2, L3) & p(L3, N) & ...
```

with two subgoals instead of one and an additional "transfer variable" `L3`.

In order to have the advantage of functional notation available PROTOS-L allows the definition of functions. Here is an example:

```
append:  list(T) x list(T) --> list(T).
    append(nil, L) = L.
    append(H.T, L) = H.append(T,L).
```

Another syntactic variant for a function definition leaves out the function name in the right hand side of the equations:

```
append:  list(T) x list(T) --> list(T).
    nil, L |> L.
    H.T, L |> H.append(T,L).
```

Thus, a function consists of a function declaration and a list of equations which may also be conditional equations, for instance:

```
number_of_cars:  list(vehicle) --> nat.
    number_of_cars(nil) = 0.
    number_of_cars(H.T) = 1 + number_of_cars(T) <-- H:car.
    number_of_cars(H.T) = number_of_cars(T). % H is not a car.
```

The equations are tried in top-down order. As soon as a left-hand-side of an equation is unifiable with the term to be rewritten and the (possibly empty) condition for that equation succeeds, the right-hand-side of the equation is taken as the value of the given function application. No other equation will then be considered. Furthermore, if at runtime none of the equations can be applied a runtime error will be generated. Thus, in this sense a function corresponds to a total deterministic relation.

The syntax for function definitions is a follows:

function_definition $\longrightarrow$

function_declaration
functional_clause
(*functional_clause*)*

function_declaration $\longrightarrow$

function_designator `':'` *domain* `'-->'` *type_term* `'.'`

functional_clause $\longrightarrow$

function_clause_head `'='` *relation_call_parameter* [`'<--'` *condition_part*] `'.'`
| {*var_term*} `'|>'` *relation_call_parameter* [`'<--'` *condition_part*] `'.'`

function_clause_head $\longrightarrow$

function_designator `'('` {*var_term*} `')'`

The conditions allowed in a function definition are exactly the same as in the definition of a relation (c.f. Section 3.2).

4.2 Built-in Functions

Most of the built-in functions are related to particular built-in types (like arithmetic types or streams) and are therefore presented in separate sections.

The built-in function

```
new_var :  T --> T.
```

serves the following purpose: Suppose we want to generate a free variable `New` with a certain type restriction at runtime, but we do not know the type restriction at compile time. Instead, we want to restrict `New` to the actual type of another variable `X`. Then

```
New = new_var(X)
```

introduces a new variable `New` whose type restriction is given by the type of `X`. The argument `X` must be a free variable.

5 Modules

In order to support modular program development with separate type checking and compilation each PROTOS-L program consists of a set of modules. Each module in turn consists of an *interface* and a *body*. As e.g. in Modula-2, the purpose of the interface of a module is to define the set of exported names which are introduced in this module or transferred from other modules. The user of a module needs only to see its interface, and not the body. A compilation unit is either an interface or a body. In order to compile such a compilation unit the interfaces of all imported modules must have been compiled, but not their bodies. The compilation of a module body of course additionally requires that its own interface has been compiled.

Besides hierarchical structuring, program bodies provide e.g. a means for the implementation of abstract data types.

Different from Modula-2, there may also be interfaces without a body, called *views* (as in TEL). The purpose of a view is to provide a name space that consists of names exported from other modules. Thus, a view is a language construct that combines a set of modules to a layer. A user of such a view must then only import this view, and has not to consider the modules behind that view.

Different from all other modules concepts, PROTOS-L provides a single type of module interface but two types of module bodies: program bodies and database bodies. Modules are the primary means for structuring PROTOS-L programs. On the interface level the structuring is transparent; thus an interface could describe both the interface to a program body or to a database body. Database bodies provide a means for structured database access, i.e. external databases can be accessed by using a database module.

In this section, only program bodies will be presented; database bodies are described in Section 6.

In the following we will first present various aspects of the module concept by working through an example, before giving the full definition of the module concept.

5.1 The Map Colouring Problem

Consider the module structure depicted in Figure 1. The top module `map` introduces predicates for colouring a given map with four colours such that no two neighbouring countries have the same colour. The definition of this top-level functionality is structured into the modules discussed in the following (for the complete example see the appendix B). We present the modules in bottom-up order and start with the module `m_country` which describes the set of countries to be coloured. Its interface

```
interface m_country.
  nmtype country := { d, f, a, cs, i, be, h, ch, l, yu, ne,
                      e, p, pl, gb }.
                  %  country will be a subtype of area in
```

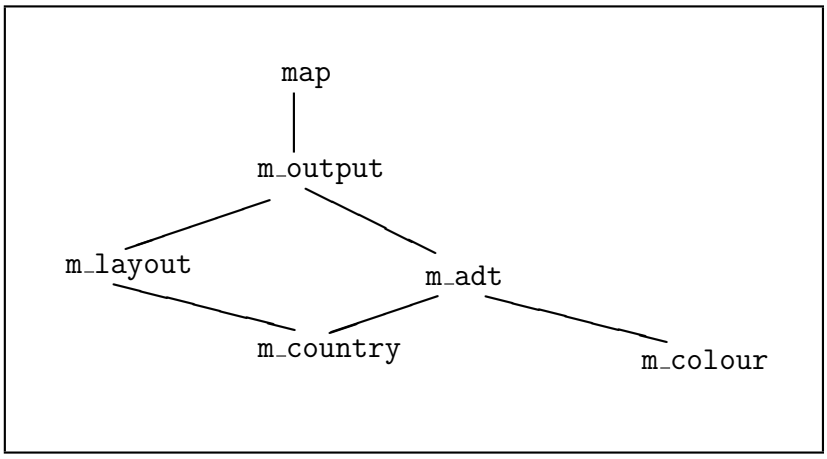


Figure 1: The module hierarchy for the map colouring problem

```
                    %  the module m_layout - thus it's an
                    %  'non-maximal type' (nmtype)
  rel all_countries : ?list(country).
                      %  yields a list of all countries
  rel all_neighbours : country x ?list(country).
                      %  for a given country yields the list
                      %  of its neighbours
endinterface.
```

introduces the type `country` which contains (most of) the countries of Europe. The relation `all_neighbours` gives for each country the list of its neighbouring countries.

The module `m_layout` provides the layout for the map to be painted. For simplicity of the presentation, we assume that we have rows of equal-sized rectangles, and every rectangle belongs to either a country or to the sea. Such a map layout can be represented by n lists (where n is the number of rows of the map visualization), each of which will be of length m (where m is the number of columns). Thus we have:

```
interface m_layout.
  imports m_country.
  from m_country : country.

  area := country ++ { s }.
  %                   sea

  rel layout : ?list(list(area)).
               % yields a simple layout for the map of Europe,
               % represented as a list of lists of area
endinterface.
```

Note that `country` can be used as a subtype of `area` since country was introduced as a non-maximal type (`nmtype`) in `m_country`. The declaration

```
from m_country : country.
```

is a *transfer declaration* stating that the type `country` is exported from `m_layout` with the definition as provided in `m_country`. This ensures that the *exported signature* of `m_layout` is self-contained in the sense that all names occurring in it have also a definition in that exported signature. If e.g. this transfer declaration was dropped from the interface, the PROTOS-L system would complain that the definition of `area` in the export signature of `m_layout` refers to a type name (`country`) that is not itself contained in exported signature.

The interface `m_colour`

```
interface m_colour.

  colour := r_y_g ++ r_y_b ++ r_g_b ++ y_g_b.

  r_y_g := r_y ++ r_g ++ y_g.
  r_y_b := r_y ++ r_b ++ y_b.
  r_g_b := r_g ++ r_b ++ g_b.
  y_g_b := y_g ++ y_b ++ g_b.

  r_y := r ++ y.
  r_g := r ++ g.
  r_b := r ++ b.
  y_g := y ++ g.
  y_b := y ++ b.
  g_b := g ++ b.

  r := { red }.
  y := { yellow }.
  g := { green }.
  b := { blue }.

  rel select_colour : ?colour.
      %  this is the non-deterministic colour choice

  rel constrain_neighbours : list(colour) x colour.
      %                      neighbours     given country

  colour_to_string : colour --> string.
      %   give a string representation for a colour

endinterface.
```

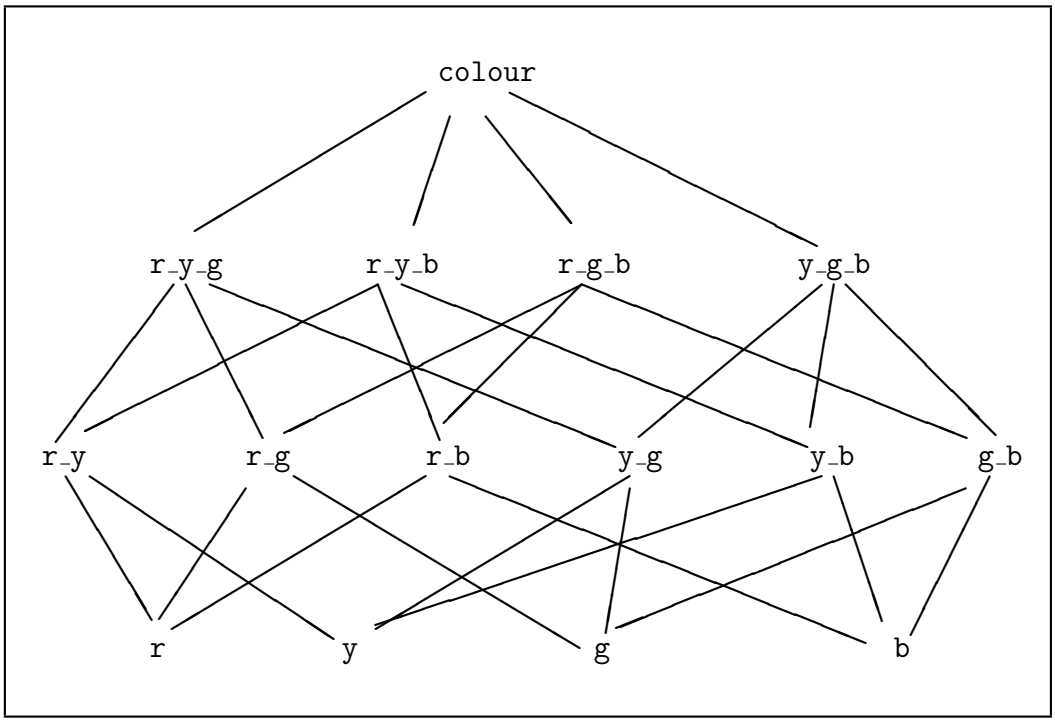


Figure 2: The colour type hierarchy for the map colouring problem

introduces a hierarchy of colour types as depicted in Figure 2.

Thus, the type `colour` represents the four possible colour values `red`, `yellow`, `green`, and `blue`. The subtypes of `colour` represent any of the possible restrictions the colour of a country must observe. For instance, if a country's colour is red then all of its neighbours must have a colour that is a subtype of `y_g_b`, and if a country has a `red` neighbour and also a `green` one then its own colour must belong to the type `y_b`.

Two relations are exported by the module `m_colour`: one for selecting a colour and another one for constraining the colours of all neighbours with respect to a given colour.

The implementation of the relations is contained in the body of `m_colour`. The implementation of `select_colour` is simply a non-deterministic relation with four ground facts:

```
rel select_colour : ?colour.
    select_colour(red).
    select_colour(yellow).
    select_colour(green).
    select_colour(blue).
```

Given any of the four colours for a country, the colour of every neighbour of it must belong to the complement type of that colour:

```
rel constrain_neighbours : list(colour) x colour.
%                          neighbours       given colour
   constrain_neighbours(nil, _).
   constrain_neighbours(Col.ColL, GivenColour) <--
           complement(GivenColour, Col) &
           constrain_neighbours(ColL, GivenColour).
```

The complement of a given colour is given by the corresponding subtype of the type `colour`:

```
rel complement : colour x ?colour.
   complement(red, Comp) <--    !Comp & Comp : y_g_b.
   complement(yellow, Comp) <-- !Comp & Comp : r_g_b.
   complement(green, Comp) <--  !Comp & Comp : r_y_b.
   complement(blue, Comp) <--   !Comp & Comp : r_y_g.
```

Note that the declaration for `complement` defines its second argument to be an output argument which should be completely instantiated after execution. However, in the complement relation the second argument is restricted to a subtype of `colour`, but it may still be a variable. Therefore, the variable `Comp` is declared as an *open variable* in the subgoal `!Comp`. The notation `!Comp` indicates to the mode checker to consider the variable as if it were bound to a ground term (c.f. Section 3.2). This achieves both: a strict mode checking discipline with enables the system to detect data flow inconsistencies while retaining the full power of Prolog's arbitrary input/output arguments in the case of open variables.

The exported function `colour_to_string` yields a string representation for each colour, to be used in the simple layout of the map:

```
colour_to_string : colour --> string.
  colour_to_string(red)    = "rr".    % in this simple version just
  colour_to_string(yellow) = "yy".    % print the first chararcter
  colour_to_string(green)  = "gg".    % of a colour twice
  colour_to_string(blue)   = "bb".    %  ...
```

Whereas the module `m_layout` implements the layout of the visualization of the map, the module `m_adt` provides a representation of the map that is more suitable for the problem solving task of colouring the map. However, the representation itself is independent of the other parts of the colouring process. Thus, these other parts should not depend on the representation; they do not have to know it. In particular, they should only use the operations that are provided on the representation to the other parts of the program. Thus, the map representation should be an abstract data type.

```
interface m_adt.

   imports m_country, m_colour, utilities.
      from m_country : country.
```

```
    from m_colour : colour,
                    r_y_g, r_y_b, r_g_b, y_g_b,
                    r_y, r_g, r_b, y_g, y_b, g_b,
                    r, y, g, b.
    from utilities: pair_of.

    map := abstract.        % abstract data type (adt).

    rel init_map : ?map.    % creates the initial map which does
                            % not yet contain a colour binding
                            % for any country

    rel set_colours : list(pair_of(country, colour)) x map.
                            % initializes the map with the colours
                            % given in the list

    rel ready : map.        % succeeds if there is no more country
                            % to be painted

    rel select_country : map x ?colour x ?list(colour) x ?map.
    %                    given            colour of
    %                    map              neighbours
    %                           colour of                  remaining
    %                           country                    map
                            % selects the next country to be painted,
                            % yields its colour restriction, the
                            % colours of its neighbours, and the
                            % remaining map still to be painted

    drel get_country_colour : map x country x ?colour.
                            % yields a country's colour

endinterface.
```

The interface `m_adt` introduces the type `map` as an *abstract* type (which is similar to an opaque type in Modula-2): No subtypes nor any constructors are given for that type. Therefore, the internal structure of the type `map` can not be investigated by another module. The only possibility for another module to use the `map` type is via the operations exported by `m_adt` (map *a*bstract *d*ata *t*ype):

```
    rel init_map : ?map.
```

creates the representation of the initial map when starting the program. The initially created map does not yet contain any bindings of a country to a particular colour.

```
    rel set_colours : list(pair_of(country, colour)) x map.
```

provides the possibility to set the colour of some countries in advance, where the polymorphic type definition `pair_of` is imported from the library module `utilities`. We will not give the full definition of that module but assume that the definition

```
pair_of( T1, T2 ) := { pair : T1 x T2 }.
```

is contained in its export interface. The relation

```
rel ready : map.
```

succeeds if there is no more country to be painted. The relation

```
rel select_country : map x ?colour x ?list(colour) x ?map.
%                    given            colour of
%                    map              neighbours
%                           colour of                  remaining
%                           country                    map
```

contains the heuristics which selects the next country to be painted (but without painting it yet). In the subgoal

```
select_country(Map, Col, ColNeigh, MapRest)
```

`Map` is the given map, `Col` is the colour variable of the selected country to be painted next, `ColNeigh` is the list of colours of its neighbours, and `MapRest` is the remaining map still to be painted.

The relation

```
drel get_country_colour : map x ?country x ?colour.
```

yields a country and its colour in a given map.

Note that the abstraction achieved indeed does not say anything about how the selection of the country is done, or how the mapping from a country to its colour or to its neighbours is realized. These things are solely defined in the module body of `m_adt` and could be changed independently from the modules using `m_adt`. (For a particular realization see the appendix).

The module `m_output` combines the information on the layout of a map and the colours selected within the abstract data type representation for maps. It produces a graphical visualization of the coloured map, together with a printed list of the countries and their colours:

```
interface m_output.
   imports m_adt.
   from m_adt: map abstract.    % map is an abstract data type
   drel print_map : map.        % shows a coloured map-picture
                                % of a part of europe on the
                                % screen and prints out the
                                % countries with its colours.
endinterface.
```

The module `map` is the top-level module of the map colouring program:

```
interface map.
   imports m_country, m_colour, utilities.
   from m_country: country.
   from m_colour : colour,
                   r_y_g, r_y_b, r_g_b, y_g_b,
                   r_y, r_g, r_b, y_g, y_b, g_b,
                   r, y, g, b.
   from utilities: pair_of.

   rel europe : list(pair_of(country,colour)).
   rel query : int.
endinterface.
```

The relation `europe` colours a map of Europe with predefined colours for the countries given in its argument list. For instance,

```
   rel europe(pair(d, blue).pair(i,blue).nil))
```

sets the colour of both Germany (`d`) and Italy (`i`) to blue before colouring the rest of Europe. The relation `query` contains some predefined queries for demonstration purposes.

The definition of `europe` initializes the map, sets the predefined colours, colours the map, and prints the result:

```
module map.
   imports m_country, m_colour, m_adt, m_output.

   rel europe : list(pair_of(country, colour)).
      europe(PairL) <--
          init_map(Map) &             % initialize map
          set_colours(PairL, Map) &   % set predefined colours
          paint(Map) &                % do the colouring
          print_map(Map).             % display the result

   rel paint : map.
```

```
        paint(Map) <--
            if ready(Map)
            then succeed
            else select_country(Map, Col, ColNeigh, Map2) &
                                          % select (variable) Col
                 select_colour(Col) &     % select value for Col
                 constrain_neighbours(ColNeigh, Col) &
                                          % neighbours of Col
                 paint(Map2)              % the rest without Col
            fi.

    rel query : int.
        query(0) <-- europe(pair(d,blue).nil).
                       % set Germany to blue
        query(1) <-- europe(pair(d,blue).pair(i,blue).nil).
                       % set Germany and Italy to blue
          ...
        query(6) <-- !DCol & DCol : g_b &
                     !FCol & FCol : g_b &
                     europe(pair(d,DCol).pair(f,FCol).nil).
                       % restrict Germany and France to green or blue
          ...
endmodule.
```

If a map `Map` has not been completely coloured yet, the relation `paint` selects (the colour variable of) a country and chooses a colour `Col` for it, constrains the colours `ColNeigh` of its neighbours to the complement of `Col` and paints the remaining part `Map2` of the map. The appendix B contains the complete map colouring program.

5.2 Module Interfaces, Bodies and Views

Here is the syntax definition for module interfaces, bodies and views. The details for database bodies are given in Section 6.

view $\longrightarrow$

‘`view`’ *module_name* ‘`.`’
‘`imports`’ {*module_name*} ‘`.`’
transfer_declaration
(*transfer_declaration*)*
‘`endview`’ ‘`.`’

interface $\longrightarrow$

program_interface | *simple_interface*

simple_interface ⟶

‘interface’ *module_name* ‘.’
(*simple_rel_declaration*)*
‘endinterface’ ‘.’

a simple interface does not allow imports or type definitions. This definition describes the subset of interfaces which can be used for database_bodies.

program_interface ⟶

‘interface’ *module_name* ‘.’
[‘imports’ {*module_name*} ‘.’
[(*transfer_declaration*)*
declaration]]
(*declaration*)*
‘endinterface’ ‘.’

body ⟶

program_body | *db_body*

program_body ⟶

‘module’ *module_name* ‘.’
[‘imports’ {*module_name*} ‘.’]
(*prog_definition*)*
‘endmodule’ ‘.’

db_body ⟶

‘database_body’ *module_name* ‘using’ {*db_name*} ‘.’
(*db_definition*)*
‘endmodule’ ‘.’

module_name ⟶

identifier

db_name ⟶

db_identifier | *path_name*

a db_identifier is the name of an AIX environment variable; thus it may also start with a capital letter
a path_name is a path name of the operating system written as a string

6 Database access in PROTOS-L

This section describes how database access is embedded in the programming language PROTOS-L, i.e. how database access is expressed within database bodies. The basic idea is to give the PROTOS-L programmer a uniform high-level database programming language. The impedance mismatch of other language integrations, e.g. the integration of SQL into C, should be avoided.

Like other implementation details access to an external database shall be transparent for the user of a module. Therefore, in an interface of a module it is not visible, whether or not the corresponding implementation of the body accesses an external database. Instead, PROTOS-L offers two kinds of module bodies: *program bodies* and *database bodies*. Program bodies support logic programming with backtracking and database bodies support access to relational databases and the definition of deductive databases, using set-oriented evaluation strategies (c.f Section 6.2).

PROTOS-L supports backtracking on top of set-oriented retrieval by allowing that program bodies import predicates from modules which are implemented by database bodies, but the database bodies can not import predicates from program bodies (and currently, also not from other database bodies).

6.1 The database module as a link to a relational database

If the implementation of a module is a database body, then there is expected to be a corresponding extensional database. In the header of the database body the logical name of the external relational database is specified. The definition of database names depends on the specific DBMS which is used. We will consider here the definition of relations which is independent from the DBMS; the technical details are given in the PROTOS-L User's Manual [Meyer and Beierle, 1994].

For every database relation declared in the database body there has to exist a corresponding database relation in the database schema of the DBMS and the types of the arguments of the declared relation have to be compatible to the types of the attributes of the underlying database relation. That is why the argument types of a declared relation are restricted to the attibute types supported by the underlying database system, i.e. to integer and string. The definition of relations of the external database, called base relations in our context, is marked with the keywords `dbrel ... is ...` In this way a PROTOS-L predicate is linked to the corresponding database relation which is described by a relation name and a sequence of attribute names. These attributes are mapped one by one onto the argument positions of the predicate. Logically, this notation can be seen as a shorthand for a sequence of all tuples of the extensional DB-relation.

Example:

```
database_body production_plan using schedule_DB .
```

```
rel    used_machines :  ?string x     ?int x     ?int x      ?int .
%                       machine name  used from  used until  for order
% Which machine is used in which time interval for which order ?
% If the machine is available in the time interval                    [2]
%            then the 4th argument has the value 0.

dbrel  used_machines    is Machine_Rel (Mname, UsedFrom, Until, Order) .


rel    step_machines :  ?int x           ?string .
%                       production step  machine name
% Which production step can be done by which machine ?

dbrel  step_machines    is Machines_for_Step (Step, Machinename) .
...

endmodule.
```

This database body requires that there are at least two relations in the database schedule_DB: `Machine_Rel` and `Machines_for_Step`. Further, `Machine_Rel` must have at least the four attributes `Mname` of type string, and `UsedFrom`, `Until` and `Order` of the type integer. Similarly, `Machine_for_Step` must have at least the attributes `Step` of the type integer and `Machinename` of type string. This is checked by the PROTOS-L system at the time when the module `production_plan` is opened. Note that the number and order of attributes of a relation can be different from the sequence of attributes listed in the database body. The latter ones only have to be a subset of the former.

6.2 The database module as a deductive database

The full power of the deductive database component appears when inference rules are defined. A PROTOS-L database body may contain function free database rules in order to implement predicates specified in the module's interface. A function free database rule consists of a head and a number of goals, each of which does not contain a function symbol. Rules in database bodies may contain the following kinds of goals:

database relation goals	e.g.	`used_machines(Machine, From, Until, Order)`
virtual relation goals	e.g.	`available_machines(Machine, F, U)`
comparison goals	e.g.	`F` $\leq$ `From`,
		`Interval` $\neq$ `2 * (Until - From)`

Note that the PROTOS-L programmer may program recursive and non-recursive rules in database bodies. This distinction is non-trivial in the field of Deductive Databases. The database body given in section 6.1 may for example additionally contain the two rules shown below. The first rule computes which machines are available in which time interval, and the second rule computes which available machines can be used for a production step in a given time interval.

[2] Note: The DB-systems don't like SQL-keywords, e.g. 'from', as attribute name.

```
database_body production_plan using schedule_DB .

...
 rel  available_machines :  ?string x     ?int x     ?int .
 %                          machine name  used from  used until
 % Which machines are available in which time interval ?

      available_machines(Machine, From, Until)
                     <--  used_machines(Machine, From, Until, 0) .


 rel  step_can_use_machine :  ?int x  ?string x     int x  int .
 %                            step    machine name  from   until
 % Which machines can be used for a production step and
 % are available in a given time interval ?

      step_can_use_machine(Step, Machine, From, Until)
                       <--  step_machines(Step, Machine) &
                            available_machines(Machine, F, U) &
                            F ≤ From &
                            U ≥ Until .

endmodule.
```

In database modules, PROTOS-L supports the built-in goals = , ≠ , ≤ , ≥ , < and > in order to compare arithmetic expressions and the built-in goals = , ≠ and "like" in order to compare strings. Arithmetic expressions may contain integer constants and variables, brackets and the operators + , – , * and // .

Since rules in program bodies and view definitions in database bodies are expressed in the same way, the PROTOS-L programmer has to learn only one single language for deductive databases and application programs. This avoids the impedance mismatch of other integrations of database query languages into host programming languages, e.g. of the integration of SQL into C.

Different from the rules in program bodies which are evaluated by backtracking, the rules in database bodies are evaluated by set-oriented query evaluation strategies. Recursive rules are evaluated by a mixed top-down and bottom-up strategy. These methods are described in [Meyer, 1989]. Set-oriented query evaluation strategies are especially advantageous if the accessed data sets are large.

6.3 Using the module concept of PROTOS-L in order to choose the evaluation strategy of rules

Whenever a rule uses facts that are stored in a database the programmer has the choice to select an adequate evaluation strategy for this rule. When the programmer assumes that there are many results of a rule R needed to solve a goal, he may prefer set-oriented evaluation of the rule R. In this case, he codes the rule R in a database body, and the

PROTOS-L system evaluates the rule set-oriented. On the other hand, if he assumes that there are only few results of a rule R needed to solve a goal, he may prefer an evaluation by backtracking and stop the evaluation when no more answers to the rule R are needed. In this case, he codes the same rule R in a program body, and the PROTOS-L system evaluates the rule by backtracking.

Example continued:

If it is assumed that in order to solve the rule `step_can_use_machine` many calls of `step_machines` are needed, then the rule `step_can_use_machine` is preferable implemented in a database body as shown in the last example, because the database body performs a set-oriented evaluation of the rule. However, if it is assumed that only a few solutions of `step_machines` is needed in order to solve the planning problem, and therefore backtracking is preferred, then the rule should be implemented in a program body instead of the database body. Hence, whether a rule accessing database relations should be implemented in a database body or in a program body depends on the desired evaluation strategy for this rule.

6.4 Integrating the knowledge of multiple databases

PROTOS-L can integrate the knowledge of many databases within a single application program. Different databases in the PROTOS-L context correspond to different areas which are under control of one database system. It is not possible to work with multiple instances of (different) database management systems. Usually every database is enclosed in its own module. The information of several databases can be integrated within program bodies that import all the predicates they need from the different database modules.

In order to provide more flexibility it is also possible to access different databases within one module. First, the '`using`' part of the module header has to contain a list of all needed databases. Second, a relation definition is augmented by '`from <dbname>`' stating that the relation is stored in the database given by `<dbname>`.

Example:

```
database_body plan2 using plantDB, localDB .

 rel    products :  ?string x     ?int .
 %                  product name  cost

 dbrel  products    is ProdRel(Name, Cost) from plantDB .


 rel    orders :  ?name x   ?string .
 %                customer  product

 dbrel  orders    is Order (Customer, ProductId) from localDB .
```

If no '`from`' part is specified the first database name from the list given in the module header is used by default.

The complete syntax of database bodies is given in Section 11. For subgoals in rule bodies it is essentially the same as in program bodies apart from the restriction to function free subgoals.

6.5 Embedded SQL

A major goal in designing the database bodies was to provide a high-level interface to relational databases which conforms to the design of program bodies. The database programming language defined up to now is independent from the specific database systems and their communication protocols. Furthermore, it does not depend on the query language of the DBMS, although only SQL-based systems are currently supported. By providing (recursive) deduction rules the computational power of the resulting language goes beyond that of standard relational query languages, e.g. SQL.

On the other hand there are some useful features in SQL like sorting and functions to compute the minimum and maximum of values which are not (yet) part of PROTOS-L. Although there are several ideas how to extend the expressive capability of inference rules we decided, (due to limited man-power) to include embedded SQL queries directly into the database language. This is for practical reasons in order to take advantage of the full power and efficiency of complex SQL queries. The syntax of the embedded SQL used in PROTOS-L is almost the same as used for other host languages.

Example:

```
rel   q :  int x ?int x ?int

      q(X,Y,Z)  <--  p1(X,Y,Z).
      q(X,Y,Z)  <--  EXEC SQL
                        SELECT MIN(R.B), MAX(R.B)
                        INTO  :2, :3
                        FROM   someRel  R
                        WHERE  R.A = :1
                     END SQL .
```

The body of any rule may be replaced by an SQL query. The query itself is parameterized by the variables occurring in the head of the clause. In the query, `X, Y, Z` are referred to by `:1, :2, :3`, respectively. At run time the current argument bindings are propagated into the SQL query which is then evaluated by the DBMS. The resulting selected values are placed into the appropriate output variables. Currently, all variables used in the WHERE part must be bound at run time and all other (output) variables have to be free. The embedded SQL query is enclosed by the keywords `EXEC SQL` and `END SQL` followed by a dot. Because SQL queries and normal deduction rules are used in the same manner we have an almost orthogonal integration.

Maybe at some time in the future PROTOS-L will allow deduction rules like

```
q(X,Y,Z)  <--  L = aggregate(A, p(A,X)) &
               Y = min(L) &
               Z = max(L).
```

replacing the SQL query shown above and thereby regaining uniformity with the other rules in database and program bodies.

7 Updatable relations and transactions

Besides the other kinds of relations (`rel, drel, trel, tdrel, proc, ...`) which may only be evaluated but may not be modified, PROTOS-L offers a special kind of *updatable relations* which may be modified.

The concept of an updatable relation (which uses the keyword `urel`) has been introduced into PROTOS-L in order to keep the logic programs readable and to avoid programming errors. Currently, *only database relations* may be updatable, i.e. an updatable relation must be implemented in a database body, and it must not be defined by rules.

```
urel   produce_on :  string x  int x  int x  string .
%                    product   from   until  machine
% Which product is produced in which time interval on which machine ?

dbrel produce_on is Production_Plan(Product, UsedFrom, Until, Machine) .
```

The facts of the updatable relation `produce_on` are taken from the external database relation `Production_Plan`. At the interface of a module it is visible which relations are updatable and which relations are not, because the exported updatable relations are declared using the keyword `urel` instead of `rel`, `drel`, `proc`, `trans`, etc.

7.1 Modification of updatable relations

PROTOS-L supports `insert` and `delete` operations on updatable relations. Insert and delete operations are only allowed on updatable relations and not on other relations. These operations are allowed only in program bodies but not in database bodies.

Insert and delete operations are embedded in the programming language as goals in program bodies, i.e. they can occur in any right hand side of a rule in a program body. The operations `insert` and `delete` are used like a meta-call. For example, in order to add the information to the database that `product1` is produced during the time interval `100 .. 200` on `machine1` , one can use the goal

```
insert produce_on("product1", 100, 200, "machine1")
```

which inserts the tuple < "product1", 100, 200, "machine1" > into the database relation `Production_Plan`. If this tuple is already present in the relation nothing has to be done. Similarly, a call of

```
delete produce_on("product1", 100, 200, "machine1")
```

deletes the tuple < "product1", 100, 200, "machine1" > from the relation `Production_Plan` if present.

The general precondition for the execution of an insert or a delete operation can be summarized as follows. Let `r` be predicate defined by a database relation `R` and $A_1, \ldots, A_n$ be arguments, i.e. A_i is either a variable or a constant. The precondition for the execution of the operations

$$\texttt{insert r}(A_1, \ldots, A_n) \quad \text{and} \quad \texttt{delete r}(A_1, \ldots, A_n)$$

is that at the calling time of the operation all arguments are bound to constants, say $c_1, \ldots, c_n$.

Insert and delete operations are always successful in PROTOS-L, independent of the tuples currently stored in the database relation R that corresponds to the predicate r. Note that this is different from retract in Prolog.

The value (the interpretation) of the modified relation after the execution of the write operation (by passing it from left to right) is the same as before the execution of the write operation with the following exception: If $\texttt{R}_{pre}$ is the value of the relation R before the execution of a write operation, then

$$\texttt{R}_{pre} \cup \{(c_1, \ldots, c_n)\}$$

is the value of the relation R after the execution of the insert operation and

$$\texttt{R}_{pre} \setminus \{(c_1, \ldots, c_n)\}$$

is the value of the relation R after the execution of the delete operation.

7.2 The integration of relation modifications and backtracking

Whenever backtracking passes a modification operation `insert` or `delete` from right to left inside a transaction, then the modification operation is undone, i.e. backtracking reestablishes the relation state given before this modification operation was executed. Hence, insert and delete operations are different from `dbassert` and `dbretract`.

The following example demonstrates how insert and delete operations on updatable relations can be used in a program part of a production planning application.

The production plan, i.e. which product is produced during which time interval on which machine, is stored in an updatable relation `produce_on`:

```
 urel produce_on :  string x  int x  int x  string .
%                   product   from   until  machine
```

An execution of the following rule changes the machine on which a product `P` is planned to be produced from machine1 to machine2, and leaves the production time interval [T1 , T2] unchanged.

```
rel change_machine :  string x int x int x string x    string .
%                     product  from  until old machine new machine

change_machine(P, T1, T2, OldM, NewM) <--
       delete produce_on(P, T1, T2, OldM) &
       insert produce_on(P, T1, T2, NewM).
```

Database modification operations may only be performed inside transactions, i.e. they may only be called when a transaction is active.

7.3 Transactions in PROTOS-L

In order to describe the scope of transactions, PROTOS-L offers the language construct transaction. A *transaction* is a special kind of a PROTOS-L relation and it is introduced by using the key word `trans` instead of `rel`. The action sequence of the transaction is programmed in the rules of the transaction.

Transactions influence the flow of program execution and backtracking.

We give the following example from our production planning application in order to demonstrate both aspects of the PROTOS-L transaction concept. A transaction `change_if_legal` changes the planned machine for producing a chemical product during the time interval [T1,T2] from machine M1 to machine M2, only if the subsequent integrity checks are successful.

```
trans change_if_legal : string x int x int x string x    string .
%                       product  from  until old machine new machine

change_if_legal(P,T1,T2,M1,M2) ←
          change_machine(P,T1,T2,M1,M2) &
          integrity_checks(P,T1,T2,M1,M2) .
```

After changing the machine the integrity checks are performed. If they are successful, i.e. if the evaluation of the rule implementing the transaction returns success, then the transaction is committed.[3] On the other hand, if the integrity checks fail, then the transaction execution can not be completed successfully and the transaction is aborted when backtracking occurs.

The integration of transactions and backtracking can be summarized and generalized as follows:

- If program execution proceeds from left to right to a goal which is implemented by a transaction, then a `BEGIN TRANSACTION` statement is performed on the underlying database system.
- If program execution returns from right to left (i.e. by backtracking with fail) from a goal which is implemented by a transaction, then the transaction is finished. Note that all modifications have been undone already during backtracking.
- If program execution proceeds from a goal implemented by a transaction from left to right, i.e. the transaction execution succeeds, then a `COMMIT TRANSACTION` statement is performed on the underlying database system.

Backtracking from right to left to a goal implemented by a transaction has to be prevented from jumping to a choice point inside the transaction, because this transaction has already been committed. Therefore, backtracking jumps to the last choice point allocated before the beginning of the transaction.

[3] At commit time the modifications of the transaction are made permanent to the database.

In order to describe the effect of transactions in more detail, we sketch a possible implementation of transactions which can be seen as a transformation where a transaction predicate

```
trans change : ...
        change(...) <--  r1
        change(...) <--  r2
```

is replaced by a top-level *deterministic* predicate containing the transaction control. The rules of the original transaction predicate are left unchanged while the predicate name is substituted by a new identifier. The auxiliary predicates ...`_transaction` will never fail. These predicates are used here for illustration purpose only and are not directly available as built-ins in PROTOS-L.

```
   drel change: ...

          change(...) <--  begin_transaction &
                           change'(...) &
                           commit_transaction.    % and succeed
                           % never coming back because
                           % change is deterministic

          change(...) <--  abort_transaction &
                           fail.

   rel change' : ...
          change'(...) <--  r1
          change'(...) <--  r2
```

Nested transaction calls are not allowed in PROTOS-L. Since the modifications of transactions are made permanent to the database at commit time, and backtracking is prevented from jumping to a choice point inside a transaction, modifications of committed PROTOS-L transactions are persistent.

For efficiency reasons PROTOS-L also provides write operations `dbassert` and `dbretract` akin to assert and retractall in Prolog. As insert and delete these operations succeed only once (note that their arguments must be ground). However, in contrast to `insert` and `delete` the effect of these operations is *not undone on backtracking.*

When allowing updates which are not undone on backtracking the question is: "Do we have a logical or an immediate update view ?"

Example:
Having the sequence

```
    p(1,Y) & ...  & dbassert p(1,99)
```

and assuming that backtracking occurs after adding the fact `p(1,99)` the flow of control may reach the subgoal `p(1,Y)` from right to left. According to the logical update view backtracking of this subgoal will not find the new fact `p(1,99)` while the immediate view defines this fact to be visible.

In PROTOS-L, currently only database relations are modified and the update view will be defined by the underlying DBMS, i.e. in general it will be the logical update view.

8 File Input and Output

In PROTOS-L file access is realized via streams. A stream represents a file that is either opened for reading or for writing. There are two types `instream(T)` and `outstream(T)` and various relations for type-safe reading and writing, see [Meyer and Beierle, 1994].

9 OSF/Motif Interface

In order to support the easy development of high-level window-based end-user systems, PROTOS-L provides a programmers interface to OSF/Motif [Jasper, 1991], [Schenk, 1991]. (Here, we will refer to AIX Windows also as OSF/Motif). The approach to integrate the PROTOS-L system with OSF/Motif is based on the PROTOS Window Manager (PWM) which resides between both systems. It supports and refines the object-oriented view of OSF/Motif that can be accessed via some built-in PROTOS-L language constructs, the so-called PWM programmer interface.

The entities manipulated by the PWM are objects of types `pwm_object` (window objects), `pwm_attribute` (attributes of window objects), and `pwm_goal` (callback routines), see [Meyer and Beierle, 1994].

Although the window interface in PROTOS-L consists of only a few built-in types and relations, it should be noted that this conceptually simple coupling opens the door to a complex and very useful system. Despite the fact that several (low-level) X-Window library and tool kit functions are not accessible in PROTOS-L we found the current capabilities sufficient to create really non-trivial user interfaces, e.g. for the knowledge based production planning system described in [Wittmann, 1991].

10 Built-ins

There are various built-in types, functions and (meta-programming) relations for

- integers,
- characters,
- bytes,
- strings,
- lists,
- arrays,
- streams,
- testing, instantiating and generating variables,

- execution control,
- the window interface,
- the C interface,
- term databases, and
- the finite domain constraint solver.

The current version of all built-ins is described in [Meyer and Beierle, 1994].

11 Syntax

- A terminal form ‘T’ means that the token T must appear physically.
- The symbol ‘|’ separates alternatives.
- An optional form [F] means that the form F is optional.
- A list form {F} means that the form F appears either once or more than once separated by commas ‘,’ .
- A star form (F)* denotes a possibly empty sequence of Fs.

Comments in PROTOS-L code start with a ’%’ and last until the end of the line.

11.1 Module Interfaces, Bodies and Views

view ⟶

‘view’ *module_name* ‘.’
‘imports’ {*module_name*} ‘.’
transfer_declaration
(*transfer_declaration*)*
‘endview’ ‘.’

interface ⟶

program_interface | *simple_interface*

simple_interface ⟶

‘interface’ *module_name* ‘.’
(*simple_rel_declaration*)*
‘endinterface’ ‘.’

a simple interface does not allow imports or type definitions. This definition describes the subset of interfaces which can be used for database_bodies.

program_interface ⟶

‘interface’ *module_name* ‘.’
[‘imports’ {*module_name*} ‘.’
[(*transfer_declaration*)*
declaration]]
(*declaration*)*
‘endinterface’ ‘.’

body ⟶

program_body | *db_body*

program_body ⟶

`'module'` *module_name* `'.'`
[`'imports'` {*module_name*} `'.'`]
(*prog_definition*)*
`'endmodule'` `'.'`

db_body ⟶

`'database_body'` *module_name* `'using'` {*db_name*} `'.'`
(*db_definition*)*
`'endmodule'` `'.'`

module_name ⟶

identifier

db_name ⟶

db_identifier | *path_name*

a db_identifier is the name of an AIX environment variable; thus it may also start with a capital letter
a path_name is a path name of the operating system written as a string

11.2 Declarations and Definitions

prog_definition ⟶

type_definition
| *type_abbreviation*
| *relation_definition*
| *function_definition*

db_definition ⟶

db_base_rel_definition | *db_virt_rel_definition*

transfer_declaration ⟶

`'from'` *module_name* `':'` {*type_or_rel_identifier*} `'.'`

declaration ⟶
type_declaration
| *relation_declaration*

type_or_rel_identifier ⟶
type_identifier [‘abstract’]
| *relation_designator*

11.2.1 Types

type_declaration ⟶
abstract_type_declaration
| *type_abbreviation*
| *type_definition*

abstract_type_declaration ⟶
mono_type_dec_lhs ‘:=’ ‘abstract’ ‘.’
| *poly_type_dec_lhs* ‘:=’ ‘abstract’ ‘.’

type_abbreviation ⟶
type_identifier ‘:=’ *ground_type_term* ‘.’
| *poly_type_dec_lhs* ‘:=’ *type_identifier* ‘(’ {*type_variable*} ‘)’ ‘.’

every variable that occurs in the left-hand side must occur in the right-hand side and vice versa

type_definition ⟶
mono_type_definition
| *poly_type_definition*

mono_type_definition ⟶
mono_type_dec_lhs ‘:=’ *mono_type_def_rhs* ‘.’

mono_type_dec_lhs ⟶
[‘nmtype’] *type_identifier*

mono_type_def_rhs ⟶

(*subtype_specification* `'++'`)*
subtype_specification `'++'`
subtype_specification
| (*subtype_specification* `'++'`)*
`'{'` {*mono_constructor_def*} `'}'`

note that in the actual PROTOS-L system there may be a restriction on the maximal number of direct subtypes

subtype_specification ⟶

type_identifier

mono_constructor_def ⟶

constr_designator [`':'` *ground_domain*]

constr_designator ⟶

identifier

ground_domain ⟶

ground_type_term [`'x'` *ground_domain*]

poly_type_definition ⟶

poly_type_dec_lhs `':='` `'{'` {*poly_constructor_definition*} `'}'` `'.'`

every variable that occurs in the left-hand side must occur in the right-hand side and vice versa

poly_type_dec_lhs ⟶

type_identifier `'('` {*type_variable*} `')'`

the occurring variables must be pairwise distinct

poly_constructor_definition ⟶

constr_designator [`':'` *domain*]

domain ⟶

type_term [`'x'` *domain*]

11.2.2 Relations

relation_definition $\longrightarrow$
relation_declaration
relational_clause
(*relational_clause*)*

relation_declaration $\longrightarrow$
rel_class relation_designator ‘:’ *io_domain* ‘.’
| ‘proc’ *relation_designator* ‘.’
| ‘tproc’ *relation_designator* ‘.’

rel_class $\longrightarrow$
‘rel’
| ‘drel’
| ‘trel’
| ‘tdrel’
| ‘proc’
| ‘tproc’
| ‘urel’
| ‘trans’

io_domain $\longrightarrow$
[‘?’] *type_term* [‘x’ *io_domain*]

relation_designator $\longrightarrow$
identifier

relational_clause $\longrightarrow$
relation_clause_head [‘<--’ *condition_part*] ‘.’

relation_clause_head $\longrightarrow$
relation_designator [‘(’ {*var_term*} ‘)’]

relation_designator $\longrightarrow$
identifier

condition_part $\longrightarrow$

condition [‘&’ *condition_part*]

condition $\longrightarrow$

relation_call
| *meta_predicate relation_call*
| *conditional*
| *variable* ‘islistof’ *var_term* ‘where’ *relation_call*
| *relation_call_parameter* ‘:’ *ground_type_term*
| *relation_call_parameter bool_op relation_call_parameter*
| ‘!’ *variable*

meta_predicate $\longrightarrow$

‘naf’
| ‘insert’
| ‘delete’
| ‘dbassert’
| ‘dbdelete’

conditional $\longrightarrow$

‘if’ *condition_part*
‘then’ *condition_part*
(‘elsif’ *condition_part* ‘then’ *condition_part*)*
[‘else’ *condition_part*]
‘fi’

relation_call $\longrightarrow$

relation_designator [‘(’ {*relation_call_parameter*} ‘)’]

11.2.3 DB relations

db_base_rel_definition $\longrightarrow$

simple_rel_declaration
‘dbrel’ *db_relation_designator* ‘is’ *db_extern_rel_declaration* ‘.’

db_extern_rel_declaration ⟶

db_extern_rel_name ‘(’ {*db_attribute_name*} ‘)’ [‘from’ *db_name*]

db_extern_rel_name and db_attribute_name are names as used in the database system; thus they may also start with a capital letter.
db_name must be one of the db_names following 'using' in that module (see Section 6.4)

db_virt_rel_definition ⟶

‘rel’ *db_relation_designator* ‘:’ *db_rel_domain* ‘.’
db_relational_clause
(*db_relational_clause*)*

simple_rel_declaration ⟶

‘rel’ *db_relation_designator* ‘:’ *db_rel_domain* ‘.’
| ‘urel’ *db_relation_designator* ‘:’ *db_rel_domain* ‘.’

db_rel_domain ⟶

[‘?’] *db_type* [‘x’ *db_rel_domain*]

db_type ⟶

‘string’ | ‘integer’

db_relational_clause ⟶

db_relation_clause_head ‘<--’ *db_condition_part* ‘.’

db_relation_clause_head ⟶

db_relation_designator ‘(’ {*db_var_term*} ‘)’

db_var_term ⟶

integer | *string* | *variable*

db_condition_part ⟶

db_condition (‘&’ *db_condition*)*
| *embedded_sql_query*

for embedded SQL queries see Section 6

db_condition ⟶

db_relation_call
| *db_relation_call_parameter db_bool_op db_relation_call_parameter*

db_relation_call_parameter is an ordinary, function-free relation_call_parameter (see 11.3), i.e. without function calls, constructors, and lists

db_relation_call ⟶

db_relation_designator `'('` {*db_relation_call_parameter*} `')'`

db_bool_op ⟶

bool_op
| `'like'`

like corresponds to the SQL operator; for bool_op see 11.3

db_relation_designator ⟶

identifier

11.2.4 Functions

function_definition ⟶

function_declaration
functional_clause
(*functional_clause*)*

function_declaration ⟶

function_designator `':'` *domain* `'-->'` *type_term* `'.'`

functional_clause ⟶

function_clause_head `'='` *relation_call_parameter* [`'<--'` *condition_part*] `'.'`
| {*var_term*} `'|>'` *relation_call_parameter* [`'<--'` *condition_part*] `'.'`

function_clause_head ⟶

function_designator `'('` {*var_term*} `')'`

11.3 Terms and Tokens

relation_call_parameter ⟶

variable
| *integer*
| *string*
| *function_call*
| *expression*
| *constr_designator* [‘(’ {*relation_call_parameter*} ‘)’]
| *relation_call_parameter* ‘.’ *relation_call_parameter*
| ‘(’ *relation_call_parameter* ‘)’

> the ‘.’ is a built-in infix-operator constructing a list; after ‘.’ there may be no blank nor any other unprintable character

var_term ⟶

variable
| *integer*
| *string*
| *constr_designator* [‘(’ {*var_term*} ‘)’]
| *var_term* ‘.’ *var_term*
| ‘(’ *var_term* ‘)’

> the ‘.’ is a built-in infix-operator constructing a list; after ‘.’ there may be no blank nor any other unprintable character

function_call ⟶

function_designator ‘(’ {*relation_call_parameter*} ‘)’

expression ⟶

relation_call_parameter expr_op relation_call_parameter

> every variable that occurs in an arithmetic expression must be bound to an integer or subtype of integer

expr_op ⟶

‘+’ | ‘-’ | ‘*’ | ‘//’ | ‘mod’

bool_op ⟶

‘=’
| ‘\=’
| *arithm_comparison*
| *string_comparison*

arithm_comparison ⟶
'=<' | '>=' | '<' | '>'

string_comparison ⟶
'@=<' | '@>=' | '@<' | '@>'

type_term ⟶
type_identifier ['(' {*var_type_term*} ')']

var_type_term ⟶
type_term | *type_variable*

ground_type_term ⟶
type_identifier ['(' {*ground_type_term*} ')']

constr_designator ⟶
identifier

function_designator ⟶
identifier

identifier ⟶
small_letter (*alpha_character*)*

no key words or built-in names (e.g. interface, database_body, put_term, put) may be used

alpha_character ⟶
letter | *digit* | '_'

integer ⟶
[~] *digit* (*digit*)*

digit ⟶
'0' ...'9'

string ⟶

‘"’ (*char*)* ‘"’

char ⟶

letter | *digit* | . . .

and symbols like `$ % & _`

type_variable ⟶

variable

variable ⟶

cap_letter (*alpha_character*)*
| *wildcard*

Note that the only allowed variable starting with '_' (wildcard) is '_' itself;
'_' may occur in longer variable names, but then not as the first character.

wildcard ⟶

‘_’

letter ⟶

cap_letter | *small_letter*

cap_letter ⟶

‘A’ ...‘Z’

small_letter ⟶

‘a’ ...‘z’

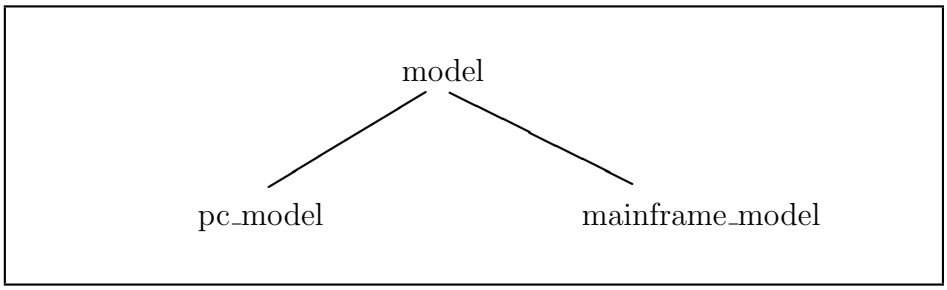


Figure 3: The computer models hierarchy

A A Job Planning Scenario

In order to further illustrate the type concept of PROTOS-L and its ability for typed unification and set-oriented answers we present a small job planning scenario as an example domain. We assume that there is a computer company offering also courses on different subjects. The employees of the company have particular qualifications, e.g. there are instructors who give courses and there are different types of technicians who can repair various kinds of computer models. The jobs to be done are given by a customer identification and an order for giving a course or for repairing some piece of hardware. A typical question to be answered by the planning system is: Who could do all jobs from a given list of jobs? The example given here is a slightly simplified version of an example developed in [Beierle and Böttcher, 1989]. Figures 3 and 4 show the subtype relationships for the types `model` and `employee` as defined in the module `jobplan`:

```
module jobplan.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                            %
% S O R T - D E F I N I T I O N S                                            %
%                                                                            %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

model           := pc_model ++ mainframe_model.
pc_model        := { pc1, pc2, pc3 }.
mainframe_model := { main1, main2, main3, main4 }.

employee             := technician ++ instructor.
technician           := pc_technician ++ mainframe_technician.
pc_technician        := allround_technician
                        ++ { peter, paul, patrick, pamela }.
mainframe_technician := allround_technician
                        ++ { mike, mary, miriam, maxwell, mark }.
allround_technician  := guru ++ { alan, adam }.
```

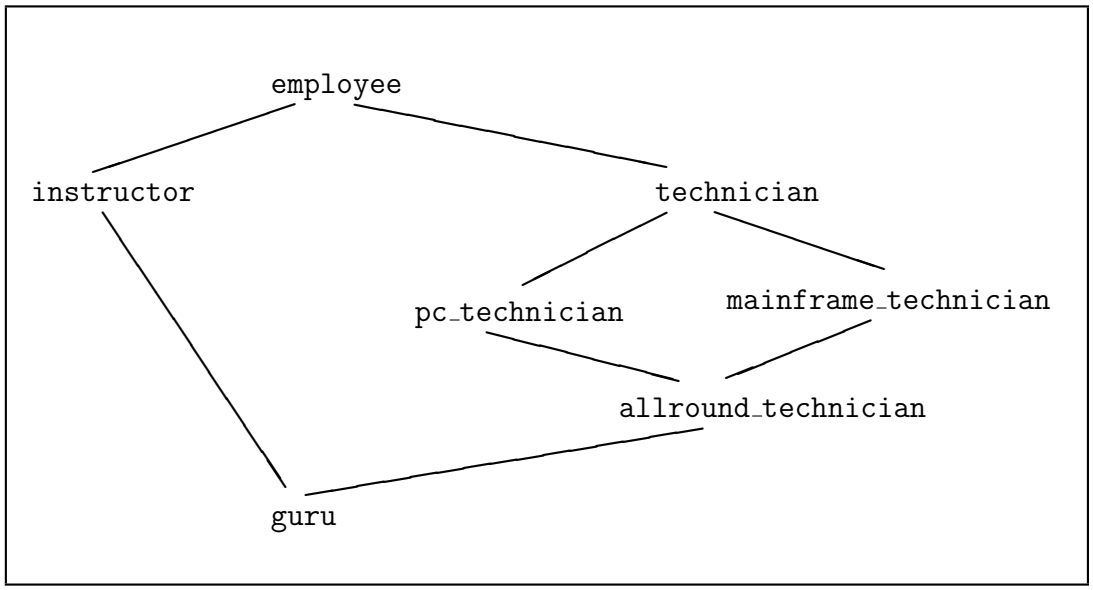


Figure 4: The subtype hierarchy for `employee`

```
instructor           := guru ++ { ingrid, ivan }.
guru                 := { george, gregor }.

course := { os2, db2, lp, xps }.

customer_id := { customer : int }.

job := { repair : customer_id x model,
         teach : customer_id x course }.



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                            %
% P R E D I C A T E S                                                        %
%                                                                            %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


rel can_repair : technician x model.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    can_repair(T, M) <-- T:pc_technician & M:pc_model.
    can_repair(T, M) <-- T:mainframe_technician & M:mainframe_model.
```

```
rel can_do_job : job x employee.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    can_do_job(repair(CID, Model), E) <-- E:technician & can_repair(E,Model).
    can_do_job(teach(CID, Course), E) <-- E:instructor.


rel can_do_all_jobs : list(job) x employee.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    can_do_all_jobs(nil,E).
    can_do_all_jobs(Job.Rest, E) <--
                            can_do_job(Job, E) &
                            can_do_all_jobs(Rest, E).


rel can_do_given_jobs : ?employee.
%%%%%%%%%%%%%%%%%%%%%%%%
    can_do_given_jobs(E) <--
             !E & can_do_all_jobs(repair(customer(290),pc2)
                                  .repair(customer(440),main1)
                                  .repair(customer(290),pc3).nil, E).

endmodule.
```

Here are some queries and the corresponding answers given by the PROTOS-L system:

```
PROTOS> can_do_given_jobs(adam).

MORE ANSWERS (Y/N)? y
    NO (MORE) ANSWERS

PROTOS> can_do_given_jobs(peter).
    NO (MORE) ANSWERS

PROTOS> can_do_given_jobs(gregor).

MORE ANSWERS (Y/N)? y
    NO (MORE) ANSWERS

PROTOS> can_do_given_jobs(E).
    E : allround_technician
MORE ANSWERS (Y/N)? y
    NO (MORE) ANSWERS

PROTOS> can_do_given_jobs(E) & E:pc_technician.
    E : allround_technician
MORE ANSWERS (Y/N)? y
```

```
    NO (MORE) ANSWERS

PROTOS> can_do_given_jobs(E) & E:instructor.
    E : guru
MORE ANSWERS (Y/N)? y
    NO (MORE) ANSWERS
```

Note that the first and the third query succeed whereas the second one fails since `peter` can not repair the mainframe `main1`. The last three queries do not instantiate the variable `E` but yield the type restrictions for `E` that represent the most general answer possible for each query.

B Map Colouring

This section contains the complete code for the map colouring problem (c.f. Section 5).

B.1 The module `map`

```
interface map.

  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  % Top module for the complete program to solve the MAP COLOURING PROBLEM.%
  %                                                                         %
  % The program uses the following modules and interfaces:                  %
  %  m_country, m_colour, m_layout, m_adt, m_output, utilities.             %
  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

  imports m_country, m_colour, utilities.
  from m_country: country.
  from m_colour : colour,
                    r_y_g, r_y_b, r_g_b, y_g_b,
                    r_y, r_g, r_b, y_g, y_b, g_b,
                    r, y, g, b.
  from utilities: pair_of.

  rel europe : list(pair_of(country,colour)).
             % colours a map of Europe with four colours; initial values may
             % be given in the parameter list (which may also be nil), e.g.
             %       europe(pair(f,blue).pair(h,blue).nil).
             % sets France and Hungary to blue

  rel query : int.                  % some predefined calls of 'europe'

endinterface.



module map.

  imports m_country, m_colour, m_adt, m_output.
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%       RELATIONS
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

rel europe : list(pair_of(country, colour)).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

europe(PairL) <--
        init_map(Map) &             % initialize map
        set_colours(PairL, Map) &   % set predefined colours
        paint(Map) &                % do the colouring
        print_map(Map).             % display the result


rel paint : map.
%%%%%%%%%%%%%%%%

paint(Map) <--
        if ready(Map)
        then succeed
        else select_country(Map, Col, ColNeigh, Map2) & % select (variable) Col
             select_colour(Col) &                       % select value for Col
             constrain_neighbours(ColNeigh, Col) &      % neighbours of Col
             paint(Map2)                                % the rest without Col
        fi.


rel query : int.
%%%%%%%%%%%%%%%%

query(0) <-- europe(pair(d,blue).nil).
                % set Germany to blue

query(1) <-- europe(pair(d,blue).pair(i,blue).nil).
                % set Germany and Italy to blue

query(2) <-- europe(pair(p,blue).pair(gb,blue).pair(f,blue).pair(ne,blue)
                    .pair(pl,blue).pair(h,blue).nil).
                % set Portugal, Great Britain, France, The Netherlands,
                % Poland, and Hungary to blue

query(3) <-- !Col & europe(pair(d,Col).pair(e,Col).pair(i,Col).nil).
                % set Germany, Spain, and Italy to the  s a m e   colour
                % (but it doesn't matter which one)

query(4) <-- europe(pair(d,blue).pair(f,blue).nil).
                % set Germany and France to blue
                % (should fail!)

query(5) <-- !Col & europe(pair(d,Col).pair(f,Col).nil).
                % set Germany and France to the  s a m e   colour
                % but it doesn't matter which one ( - should also fail!)

query(6) <-- !DCol & DCol : g_b &
```

```
               !FCol & FCol : g_b &
               europe(pair(d,DCol).pair(f,FCol).nil).
                  % restrict Germany and France to green or blue
                  % (should succeed!)

endmodule.
```

B.2 The module m_output

```
interface m_output.

  imports m_adt.
  from m_adt: map abstract.         % map is an abstract data type

  drel print_map : map.   % shows a coloured map-picture of a part of europe
                          % on the screen and prints out the countries with
                          % its colours.

endinterface.



module m_output.

  imports m_country, m_colour, m_layout, m_adt.

  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  %
  %      RELATIONS
  %
  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

  drel print_map : map.     % shows a coloured map-picture of a part of europe
  %%%%%%%%%%%%%%%%%%%%      % on the screen and prints out the countries with its
                            % colours.
  print_map(Map) <--
          layout(MapLayout) &                  % get the layout
          print_map_picture(Map, MapLayout) &   % print the map
          nl &
          all_countries(Countries) &           % get list of countries
          print_map_colours(Countries, Map, 0). % .. and print their colours


  drel print_map_picture : map x list(list(area)).
  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

  print_map_picture(_, nil).                   % print each line
  print_map_picture(Map, HMapC.TMapC) <--      % of the map
          print_map_line(Map, HMapC) &
          print_map_picture(Map, TMapC).


  drel print_map_line : map x list(area).
  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

  print_map_line(_, nil) <-- nl.
```

```
  print_map_line(Map, Area.Tail) <--
          if Area = s
             then put_string(user_output, "  ")         % sea background
             else
                Area:country &
                get_country_colour(Map, Area, Colour) & % get its colour
                CS = colour_to_string(Colour) &         % .. and colour string
                put_string(user_output, CS)             % and write it
          fi  &
          print_map_line(Map, Tail).


  drel print_map_colours : list(country) x map x int.
  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

  print_map_colours(nil, _ ,_).
  print_map_colours(Country.CL, Map, No) <--
          get_country_colour(Map, Country, Col) &
          write(Country) &
          write_string(" : ") &
          write(Col) &
          write_string("    ") &
          M = No+1 &                                % after every five
          if M mod 5 = 0 then nl fi &               % countries start a nl
          print_map_colours(CL, Map, M).

endmodule.
```

B.3 The module m_layout

```
interface m_layout.

  imports m_country.
  from m_country : country.

  area := country ++ { s }.
  %                    sea

  rel layout : ?list(list(area)).
              % yields a simple layout for the map of Europe,
              % represented as a list of lists of area

endinterface.



module m_layout.

  imports m_country.

  area := country ++ { s }.
            %         sea

  rel layout : ?list(list(area)).
              % yields a simple layout for the map of Europe,
```

```
            % represented as a list of lists of area
  layout(Layout) <-- Layout =
     (s .s .s .gb.s .s .s .s .s .s .s .s .s .s .s .s .s .s .nil)
    .(s .s .gb.gb.s .s .s .s .s .s .s .s .s .s .s .s .s .s .nil)
    .(s .s .gb.gb.s .s .s .s .s .s .s .s .s .s .s .s .s .s .nil)
    .(s .s .s .gb.gb.s .s .s .s .s .d .s .s .s .s .pl.pl.s .s .s .nil)
    .(s .s .s .gb.gb.gb.s .s .ne.d .d .d .s .s .pl.pl.pl.pl.s .s .nil)
    .(s .s .s .gb.gb.gb.s .ne.ne.d .d .d .d .pl.pl.pl.pl.pl.s .s .nil)
    .(s .s .gb.gb.gb.gb.s .be.be.d .d .d .d .pl.pl.pl.pl.pl.s .s .nil)
    .(s .s .s .s .s .s .s .f .l .d .d .d .d .cs.cs.cs.pl.s .s .s .nil)
    .(s .s .s .s .f .f .f .f .f .d .d .d .d .a .cs.cs.cs.cs.s .s .nil)
    .(s .s .s .s .s .f .f .f .f .ch.ch.a .a .a .h .h .h .h .s .s .nil)
    .(s .s .s .s .s .f .f .f .f .i .i .i .i .yu.yu.h .h .h .s .s .nil)
    .(s .s .s .s .s .f .f .f .f .i .i .i .i .s .yu.yu.yu.yu.s .s .nil)
    .(s .s .e .e .e .e .e .f .s .s .s .i .i .s .s .yu.yu.yu.s .s .nil)
    .(s .s .p .e .e .e .e .e .s .f .s .s .i .i .s .s .s .s .s .s .nil)
    .(s .s .p .e .e .e .e .s .s .s .i .s .s .s .i .i .s .s .s .s .nil)
    .(s .s .p .e .e .e .e .s .s .s .s .s .s .s .i .s .s .s .s .s .nil)
    .(s .s .p .e .e .e .s .s .s .s .s .s .i .i .s .s .s .s .s .s .nil)
    .nil.

endmodule.
```

B.4 The module m_adt

```
interface m_adt.

  imports m_country, m_colour, utilities.
  from m_country : country.
  from m_colour : colour,
                  r_y_g, r_y_b, r_g_b, y_g_b,
                  r_y, r_g, r_b, y_g, y_b, g_b,
                  r, y, g, b.
  from utilities: pair_of.

  map := abstract.        % abstract data type (adt).

  rel init_map : ?map.    % creates the initial map which does
                          % not yet contain a colour binding
                          % for any country

  rel set_colours : list(pair_of(country, colour)) x map.
                          % initializes the map with the colours
                          % given in the list

  rel ready : map.        % succeeds if there is no more country
                          % to be painted

  rel select_country : map x ?colour x ?list(colour) x ?map.
  %                    given            colour of
  %                    map              neighbours
  %                           colour of                  remaining
  %                           country                    map
                          % selects the next country to be painted,
                          % yields its colour restriction, the
```

```
                          % colours of its neighbours, and the
                          % remaining map still to be painted

  drel get_country_colour : map x country x ?colour.
                          % yields a country's colour

endinterface.


module m_adt.

  imports m_country, m_colour.

  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  %
  %      DEFINITIONS
  %
  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

  map := list(map_entry).                 % defines a representation for
                                          % the abstract type map

  map_entry := { entry : country x          % the country
                         colour x           % its colour
                         list(colour) x     % colours of its neighbours
                         list(map_entry) }. % entries of its neighbours

  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  %
  %      RELATIONS
  %
  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

  rel init_map : ?map.                    % creates the initial map
  %%%%%%%%%%%%%%%%%%%

  init_map(Map) <--
          all_countries(CountL) &         % get list of all countries
          create_entries(CountL, Map) &   % create map entries for all of them
          connect_entries(Map, Map).      % .. and connect the entries


  rel create_entries : list(country) x ?map.
  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  % creates the map-entries with open variables for colour- and neighbour-lists

  create_entries(nil, nil).
  create_entries(Country.CountryList,
                 entry(Country, Colour, ColourList, EntryList).Map) <--
          create_entries(CountryList, Map) &
          !Colour &           %  its colour variable
          !ColourList &       %  list of colours of its neighbours
          !EntryList.         %  list of entries of its neighbours
                              %  ... these three variables are still free here;
                              %  they must be declared as 'open' variables
                              %  since they occur in an output position
```

```
rel connect_entries : map x map.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% initializes the 'colour' and 'entry' list-variables for every country

connect_entries(nil, Map).
connect_entries(entry(Count, _, ColL, EntL).MapRest, Map) <--
        all_neighbours(Count, NeighbL) &      %  this will generate
        connect(NeighbL, ColL, EntL, Map) &   %  a cyclic structure
        connect_entries(MapRest, Map).        %  for map !


rel connect : list(country) x list(colour) x list(map_entry) x map.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% generates the list of variables for the neighbours of a country

connect(nil, nil, nil, Map).
connect(Count.CountL, Col.ColL, Ent.EntL, Map) <--
        get_entry(Count, Map, Ent) &
        entry(_, Col, _, _) = Ent &
        connect(CountL, ColL, EntL, Map).


drel get_entry : country x map x ?map_entry.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% yields the entry for a given country of the map

get_entry(Count, Ent.Map, Ent) <--
        entry(Count, _, _, _) = Ent.
get_entry(Count, Ent1.Map, Ent2) <--
        get_entry(Count, Map, Ent2).


rel ready : map.
%%%%%%%%%%%%%%%%%%%%
% succeeds, if there is no more country to be painted

ready(nil).


rel select_country : map x ?colour x ?list(colour) x ?map.
%                    given           colour of
%                    map             neighbours
%                           colour of                 remaining
%                           country                   map
% This relation contains the heuristic which, from the given map,
% selects the next country to be painted, yields its colour restriction,
% the colours of its neighbours, and the remaining map still to be
% painted.
% In this case the selected country is simply the first one in the
% map representation

select_country(entry(Count, Col, ColL, _).Map, Col, ColL, Map).


rel set_colours : list(pair_of(country, colour)) x map.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
  % initializes the map with the colours given in the list

  set_colours(nil, _).
  set_colours(pair(Count, Col).PairL, Map) <--
          get_entry(Count, Map, entry(Count, Col, ColL, _)) &
          if var(Col)                           % if Col is not a
             then succeed                       % variable, constrain
             else constrain_neighbours(ColL, Col) % the neighbour's colours
          fi &
          set_colours(PairL, Map).


  drel get_country_colour : map x country x ?colour.
  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  % yields a country's colour

  get_country_colour(entry(Count, Col, _, _)._, Count, Col).
  get_country_colour(_.Map, Count, Col) <--
          get_country_colour(Map, Count, Col).

endmodule.
```

B.5 The module m_country

```
interface m_country.

  nmtype country := { d, f, a, cs, i, be, h, ch, l, yu, ne, e, p, pl, gb }.
                    %  country will be a subtype of area in the module
                    %  m_layout - thus it's an 'non-maximal type' (nmtype)

  rel all_countries : ?list(country).
                      % yields a list of all countries

  rel all_neighbours : country x ?list(country).
                      % for a given country yields the list of its neighbours

endinterface.



module m_country.

  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  %
  %      DEFINITIONS
  %
  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

  nmtype country := { d, f, a, cs, i, be, h, ch, l, yu, ne, e, p, pl, gb }.

  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  %
  %      RELATIONS
  %
  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
  rel all_countries : ?list(country).
  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

  all_countries(L) <-- !C & C:country & instantiate(C, L).


  rel all_neighbours : country x ?list(country).
  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

  all_neighbours(d,  ne.be.l.f.ch.a.cs.pl.nil).
  all_neighbours(f,  e.be.l.d.ch.i.nil).
  all_neighbours(cs, d.a.h.pl.nil).
  all_neighbours(a,  d.ch.i.yu.cs.h.nil).
  all_neighbours(i,  f.ch.a.yu.nil).
  all_neighbours(be, ne.f.l.d.nil).
  all_neighbours(h,  cs.a.yu.nil).
  all_neighbours(ch, d.f.i.a.nil).
  all_neighbours(l,  be.f.d.nil).
  all_neighbours(yu, i.a.h.nil).
  all_neighbours(ne, be.d.nil).
  all_neighbours(e,  p.f.nil).
  all_neighbours(p,  e.nil).
  all_neighbours(pl, d.cs.nil).
  all_neighbours(gb, nil).

endmodule.
```

B.6 The module m_colour

```
interface m_colour.

  colour := r_y_g ++ r_y_b ++ r_g_b ++ y_g_b.

  r_y_g := r_y ++ r_g ++ y_g.
  r_y_b := r_y ++ r_b ++ y_b.
  r_g_b := r_g ++ r_b ++ g_b.
  y_g_b := y_g ++ y_b ++ g_b.

  r_y := r ++ y.
  r_g := r ++ g.
  r_b := r ++ b.
  y_g := y ++ g.
  y_b := y ++ b.
  g_b := g ++ b.

  r := { red }.
  y := { yellow }.
  g := { green }.
  b := { blue }.

  rel select_colour : ?colour.
      %  this is the non-deterministic colour choice

  rel constrain_neighbours : list(colour) x colour.
```

```
      %                       neighbours     given country

  colour_to_string : colour --> string.
      %   give a string representation for a colour

endinterface.


module m_colour.

  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  %
  %      TYPE DEFINITIONS
  %
  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

  colour := r_y_g ++ r_y_b ++ r_g_b ++ y_g_b.

  r_y_g := r_y ++ r_g ++ y_g.
  r_y_b := r_y ++ r_b ++ y_b.
  r_g_b := r_g ++ r_b ++ g_b.
  y_g_b := y_g ++ y_b ++ g_b.

  r_y := r ++ y.
  r_g := r ++ g.
  r_b := r ++ b.
  y_g := y ++ g.
  y_b := y ++ b.
  g_b := g ++ b.

  r := { red }.
  y := { yellow }.
  g := { green }.
  b := { blue }.

  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  %
  %      RELATIONS
  %
  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

  rel select_colour : ?colour.
  %%%%%%%%%%%%%%%%%%%%%%%%%%%%

  select_colour(red).
  select_colour(yellow).
  select_colour(green).
  select_colour(blue).


  rel complement : colour x ?colour.
  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

  complement(red, Comp) <--    !Comp & Comp : y_g_b.
  complement(yellow, Comp) <-- !Comp & Comp : r_g_b.
  complement(green, Comp) <--  !Comp & Comp : r_y_b.
  complement(blue, Comp) <--   !Comp & Comp : r_y_g.
```

```
  rel constrain_neighbours : list(colour) x colour.
  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

  constrain_neighbours(nil, _).
  constrain_neighbours(Col.ColL, GivenColour) <--
          complement(GivenColour, Col) &
          constrain_neighbours(ColL, GivenColour).


  colour_to_string : colour --> string.
  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

  colour_to_string(red)    = "rr".    % in this simple version just
  colour_to_string(yellow) = "yy".    % print the first chararcter
  colour_to_string(green)  = "gg".    % of a colour twice
  colour_to_string(blue)   = "bb".    %  ...
                                      % another simple alternative would
                                      % be to use escape characters, e.g.:
  % colour_to_string(red)    = S <-- C=nat_to_char(219) &
  %                                  S = list_to_string(C.C.nil).
  % colour_to_string(yellow) = S <-- C=nat_to_char(176) &
  %                                  S = list_to_string(C.C.nil).
  % colour_to_string(green)  = S <-- C=nat_to_char(177) &
  %                                  S = list_to_string(C.C.nil).
  % colour_to_string(blue)   = S <-- C=nat_to_char(178) &
  %                                  S = list_to_string(C.C.nil).

endmodule.
```

B.7 The module `utilities`

The module `utilities` is not given completely here; we only assume that its interface contains the polymorphic type definition for pairs used above:

```
interface utilities.

  pair_of(T1, T2) := {pair : T1 x T2}.

  ...

endinterface.
```

References

[Bancilhon and Ramakrishnan, 1986] Francois Bancilhon and Raghu Ramakrishnan. An Amateur's Introduction to Recursive Query Processing Strategies. In *Int. Conf. on Management of Data, ACM SIGMOD*, Washington D.C., May 1986.

[Bancilhon *et al.*, 1986] F. Bancilhon, D. Maier, Y. Sagiv, and J.D. Ullman. Magic Sets and other strange ways to implement logic programs. In *Proc. ACM SIGACT-SIGMOD Symp. on Principles of Database Systems*, 1986.

[Bancilhon, 1986] Francois Bancilhon. Naive Evaluation of Recursive Defined Relations. In Brodie and Mylopoulos, editors, *On Knowledge Base Management Systems (topics in information systems)*. Springer, 1986.

[Bayer, 1985] Rudolf Bayer. Database Technology for Expert Systems. In *GI-Kongress: Wissensbasierte Systeme*, München, 1985. Inf. FB 112, Springer.

[Beierle and Börger, 1992] C. Beierle and E. Börger. Correctness proof for the WAM with types. In E. Börger, H. Kleine Büning, G. Jäger, and M. M. Richter, editors, *Computer Science Logic - CSL'91*, volume 626 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1992.

[Beierle and Böttcher, 1989] C. Beierle and S. Böttcher. PROTOS-L: Towards a knowledge base programming language. In W. Brauer and C. Freksa, editors, *Proceedings GI-Kongreß Wissensbasierte Systeme*. Springer-Verlag, 1989.

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

[Beierle *et al.*, 1989] C. Beierle, J. Dörre, U. Pletat, C.-R. Rollinger, and R. Studer. The knowledge representation language L-LILOG. In E. Börger, H. Kleine Büning, and M. M. Richter, editors, *CSL'88 - 2nd Workshop on Computer Science Logic*, volume 385 of *Lecture Notes in Computer Science*, pages 14–51. Springer-Verlag, Berlin, 1989.

[Beierle *et al.*, 1991a] C. Beierle, S. Böttcher, and G. Meyer. Draft report of the logic programming language PROTOS-L. IWBS Report 175, IBM Germany, Scientific Center, Inst. for Knowledge Based Systems, Stuttgart, 1991.

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

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

[Beierle, 1992] 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.

[Beringer and De Backer, 1994] H. Beringer and B. De Backer. Combinatorial problem solving in constraint logic programming with cooperating solvers. In C. Beierle and L. Plümer, editors, *Logic Programming: Formal Methods and Practical Applications*, Studies in Computer Science and Artificial Intelligence. North-Holland, 1994. (to appear).

[Bocca, 1986] J. Bocca. On the evaluation strategy of EDUCE. In *Proc. ACM-SIGMOD Conference*, Washington, D.C., 1986.

[Böttcher and Beierle, 1989] S. Böttcher and C. Beierle. Data base support for the PROTOS-L system. *Microprocessing and Microprogramming*, 27, August 1989.

[Böttcher, 1990] S. Böttcher. How to use PROTOS-L as a logic-based database programming language. In *The EUREKA Project PROTOS*, 1990.

[Ceri *et al.*, 1990] S. Ceri, G. Gottlob, and L. Tanca. *Logic Programming and Databases*. Surveys in Computer Science. Springer Verlag, 1990.

[DeGroot and Lindstrom, 1986] D. DeGroot and G. Lindstrom, editors. *Functional and Logic Programming*. Prentice Hall, 1986.

[Dietrich and Hagl, 1988] R. Dietrich and F. Hagl. A polymorphic type system with subtypes for Prolog. In *Proceedings of the 2nd European Symposium on Programming*, Lecture Notes in Computer Science, pages 79–93, Berlin, 1988. Springer-Verlag.

[Futatsugi *et al.*, 1985] K. Futatsugi, J. Goguen, J.-P. Jouannaud, and J. Meseguer. Principles of OBJ2. In B. Reid, editor, *Proceedings of 12th ACM Conference on Principles of Programming Languages*, pages 52–66. ACM, 1985.

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

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

[Hanus, 1989] M. Hanus. Horn clause programs with polymorphic types: Semantics and resolution. In *Proceedings TAPSOFT'89*. Springer-Verlag, 1989.

[Harper *et al.*, 1986] R. Harper, D. MacQueen, and R. Milner. Standard ML. Report ECS-LFCS-86-2, Dep. of Computer Science, Univ. of Edinburgh, 1986.

[Hauner, 1989] I. Hauner. Database access for PROTOS-L. Diplomarbeit, EWH Koblenz und IBM Deutschland GmbH, Stuttgart, December 1989. (in German).

[Hill and Lloyd, 1992] P. M. Hill and J. W. Lloyd. The Gödel Report. TR-91-02, Dept. of Computer Science, University of Bristol, Bristol, UK, Revised Version, June 1992.

[Huber and Varsek, 1987] M. Huber and I. Varsek. Extended Prolog for order-sorted resolution. In *Proceedings of the 4th IEEE Symposium on Logic Programming*, pages 34–45, San Francisco, 1987.

[Hulin, 1989] Guy Hulin. Parallel Processing of Recursive Queries in Distributed Architectures. In *Proc. 15th Int. Conf. on VLDB*, Amsterdam, 1989.

[IBM, 1989] IBM. *IBM Prolog for 370: Language Reference*. International Business Machines Corporation, 1989.

[Jasper, 1991] H. Jasper. A logic-based programming environment for interactive applications. In *Proc. Human Computer Interaction International*, Stuttgart, 1991.

[Meyer and Beierle, 1994] G. Meyer and C. Beierle. PROTOS-L Users's Manual. Working Paper No 5, IBM Germany, Scientific Center, Inst. for Logics and Linguistics, Heidelberg, July 1994.

[Meyer *et al.*, 1994] G. Meyer, C. Beierle, and R. Scheubrein. Aspects of coupling logic programming and databases. In H. H. Bock, W. Lenski, and M. M. Richter, editors, *Information systems and data analysis*, volume 4 of *Studies in Classification, Data Analysis, and Knowledge Organization*. Springer-Verlag, 1994. (to appear).

[Meyer, 1989] G. Meyer. Rule evaluation on databases in the PROTOS-L system. Diplomarbeit Nr. 630, Universität Stuttgart and IBM Deutschland GmbH, Stuttgart, December 1989. (in German).

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

[Nutt and Smolka, 1993] W. Nutt and G. Smolka. Implementing TEL. SEKI-Report, FB Informatik, Universität Kaiserslautern, 1993. (in preparation).

[Pletat and v. Luck, 1990] U. Pletat and K. v. Luck. Knowledge representation in LILOG. In K. H. Bläsius, U. Hedtstück, and C.-R. Rollinger, editors, *Sorts and Types for Artificial Intelligence*, volume 418 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, Berlin, 1990.

[Quintus, 1987] Quintus. *Quintus Prolog Data Base Interface Manual*. Quintus Computer Systems Inc., California, 1987.

[Schenk, 1991] M. Schenk. Graphical user interface for the PROTOS-L system. Diplomarbeit Nr. 763, Universität Stuttgart und IBM Deutschland GmbH, Stuttgart, January 1991. (in German).

[Semle, 1989] H. Semle. Extension of an abstract machine for order-sorted prolog to polymorphism. Diplomarbeit Nr. 583, Universität Stuttgart and IBM Deutschland GmbH, Stuttgart, April 1989. (in German).

[Smolka, 1988a] G. Smolka. Logic programming with polymorphically order-sorted types. In J. Grabowski, P. Lescanne, and W. Wechler, editors, *Algebraic and Logic Programming*, Berlin, 1988. Akademie-Verlag.

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

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

[Vassiliou, 1986] Yannis Vassiliou. Knowledge Based and Database Systems: Enhancements, Coupling or Integration. In Brodie and Mylopoulos, editors, *On Knowledge Base Management Systems (topics in information systems)*. Springer, 1986.

[Vieille, 1988] Laurent Vieille. From QSQ towards QoSaQ: Global Optimization of Recursive Queries. In *Proc. 2nd Int. Conf. on Expert Database Systems*, Virginia, April 1988.

[Walther, 1985] C. Walther. A mechanical solution of Schubert's steamroller by many-sorted resolution. *Artificial Intelligence*, 26:217–224, 1985.

[Walther, 1988] C. Walther. Many-sorted unification. *Journal of the ACM*, 35(1):1–17, January 1988.

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

[Wittmann, 1991] H. Wittmann. An example for knowledge based production planning with PROTOS-L. Diplomarbeit, Universität Stuttgart und IBM Deutschland GmbH, Stuttgart, 1991. (in German).

[Zeller, 1990] M. Zeller. Extension of a compiler for PROTOS-L by a module concept. Diplomarbeit, Universität Stuttgart und IBM Deutschland GmbH, Stuttgart, January 1990. (in German).