facultad de informática

universidad politécnica de madrid

A Proposal for an Interchange Abstract Syntax for (Parallel) Prolog

Francisco Bueno, Manuel Carro,
Daniel Cabeza, Francisco Ballesteros,
Pedro López García,
María José García de la Banda,
Manuel Hermenegildo, Luis Gómez,
Steven Prestwich and Shan–When Yan

A Proposal for an Interchange Abstract Syntax for (Parallel) Prolog

Authors

Francisco Ballesteros, Francisco Bueno, Daniel Cabeza, Manuel Carro, María José García de la Banda, Luis Gómez, Manuel Hermenegildo, Pedro López García.
Universidad Politécnica de Madrid (UPM), Facultad de Informática,
28660–Boadilla del Monte, Madrid — Spain.
{nemo,bueno,bardo,boris,maria,lgomez,herme,pedro}@dia.fi.upm.es

Steven Prestwich, Shan–When Yan.
European Computer–industry Research Center (ECRC), Arabellastraβe, 17
8000 München 81 — Germany.
{Steven.Prestwich,swyan}@ecrc.de

Abstract

We propose an abstract syntax for Prolog that will help the manipulation of programs at compile–time, as well as the exchange of sources and information among the tools designed for this manipulation. This includes analysers, partial evaluators, and program transformation tools. We have chosen to concentrate on the information exchange format, rather than on the syntax of programs, for which we assume a simplified format. Our purpose is to provide a low–level meeting point for the tools which will allow them to read the same programs and understand the information about them. This report describes our first design in an informal way. We expect this design to evolve and concretize, along with the future development of the tools, during the project.

1 Introduction

A meaningful task in the process of program compilation is that of program analysis. Any program analysis, be it local or global, aims at inferring information about the run–time properties of the program. The knowledge of this at compile–time can be of great usefulness for the various tasks a compiler may perform. In any case, such information could be provided by the user him/herself, and thus the purpose of the compile–time analysis can be viewed as to relieve the programmer of such a burden.

On the other hand, we believe that the counterpart objective must also be pursued. Thus, it should be possible for the programmer to provide any information an analysis can give, and that the compiler (and the language) should provide means to do this. In this spirit, the declarations herein presented can be viewed both as internal representations for the information gathered by the different compilation tools and as user–level declarations that he/she can arbitrarily dispose to annotate his/her program.

In this line, such declarations serve the two main purposes for which they have been conceived: to give means for (maybe very) different compile–time tools to communicate knowledge about the program between them, and to allow the user to facilitate the task of these tools by providing them with additional information.

Nonetheless, it can be argued that the declarations as defined are too low–level to be easily manipulated by the user. Besides, the user may want to define and make explicit some knowledge about the program that is too high–level to be used directly by the compilers. Thus, some "syntactic sugar" may be defined for this purpose. This is the case of some declarations which act as a higher level encoding of others (cf. the `mode/2` and `data/2` declarations presented below), or of some predefined *macros* introduced in the last section of this report.

There are a number of other declarations which alter the program semantics and/or behaviour, instead of simply declaring properties of the program (although they also state properties that hold in this altered semantics). These declarations are needed if that specific semantics is pursued. This is the case of suspensions or dynamic predicates (and of modules, although these are better seen as a guide to the compiler than as a modification of the program execution).

On the other hand, most declarations of properties are optional for the programmer. But in certain cases (meta- and dynamic predicates are the best examples) some compilation tools, as global analysis based on abstract interpretation, can not proceed with the required accuracy. Thus, in these cases the user should be warned that no refinement of his/her program can be done unless he/she supplies the required information.

2 Notation

In this proposal we will use the following notation, which is also the most widely recognized as a standard one:

- The *predicate spec* for the predicate whose name is `predicate_name` and whose arity is `n` is `predicate_name/n`.
- The declaration

  ```
  :- goal_1, ...  , goal_n
  ```

  is a *directive* which corresponds to the idea of a *query*. When placed in a file, `goal_1, ..., goal_n` are executed when the file is read in, be it for consulting or compiling.

  The order in which directives are executed is not determined within a file (this happens in SICStus [Car88]), but when a file has been consulted, all of its directives have been executed.
- A *most general goal pattern* (or simply "goal pattern," hereafter) is a *normalized* pattern for goals of a predicate, i.e. a goal where all arguments are distinct variables.

A world–wide recognized standard for Prolog syntax is that of Edinburgh Prolog. In this report, we adopt this syntax, but disregarding any "syntactic sugar" which could be added to it. This allows meeting different program manipulation tools at low–level "plain" syntax, which can be obtained by a simple pre–processing of the programs. The program syntax for our standard is thus defined by a simple grammar allowing for (sequential and parallel) conjunctions of atoms, some operators, and annotations:

$Program ::= Declaration.Program \mid Declaration.$
$Declaration ::= Clause \mid :- Annotation$
$Clause ::= atom :- Body \mid atom$
$Body ::= Goal,\ Goal \mid Goal$
$Goal ::= Body\ \&\ Body \mid ! \mid pragma \mid Op\ atom \mid atom$
$Op ::= \backslash +$
$Annotation ::= entry \mid exit \mid others$

This syntax does not include at this point goals in directives, as explained above. Instead, directives are used only for annotations. The format of these annotations (*pragma*, *entry*, *exit* and *others*) is the subject of the rest of the report.

Annotations in a program are a way to instruct the compiler to generate better code (in terms of general efficiency) while retaining the semantics of the program without such annotations, and to find possible programming mistakes. They can be explicitly written by the programmer or generated by a compiler. If supplied by the programmer, it would be desirable that the compiler check them in order to verify their correctness.

Annotations are to be regarded as *assertions* over the program: they state a property which holds under the conditions for which these assertions were generated. In this sense they are semantically different from directives (albeit probably syntactically equally formed) because they do not change the meaning of the program, neither are they executed in any way. On the other hand, if properly generated, annotations reflect the program meaning.

Annotations can appear at two levels:

- Predicate level.
- Goal level.

Annotations at the *goal level* refer to the state of the variables of the clause just at the point where the annotation appears: between two goals, after the head of a clause or after the last goal of a clause. We propose reserving the literal `pragma/1` (as in [oBLdM92]) to enclose all necessary information at a given point in the execution:

```
:- ..., goal_1, pragma([mode(...),type(...),...]), goal_2, ...
```

where the *pragma* info is valid before calling `goal_2` and also after calling `goal_1`, that is, at the exit point for `goal_1` and at the entry point of `goal_2`. The intended meaning of `pragma/1` as a part of the program is the one given by the following definition:

```
pragma(_).
```

The information given by `pragma/1` can refer to any of the variables in the clause. The type of information that can be expressed is augmented just by adding more functors to the list `pragma/1` declares.

Information at the predicate–level is valid for any occurrence of the affected predicate as a goal in the program. Information at this level can be specified using a directive style syntax. Predicate–level annotations refer to the state of the variables appearing as arguments, right before entry and/or after exit of any call to the predicate, and can be seen as `pragma/1` annotations which affect all the occurrences of the literal as a goal. Under this point of view, predicate level annotations summarize a set of goal level annotations.

We propose to use two only predicate–level declarations: `entry/2` and `exit/2`, which will specify the information which is valid at the entry point and at the exit point, respectively, of calls to the given predicate. For example:

```
:- entry(goal_pattern,[mode(...),type(...),...]).

:- exit(goal_pattern,[mode(...),type(...),...]).
```

Extensions to the type of information provided by `entry/2` and `exit/2` declarations are made in much the same way as with `pragma/1`. Thus we define the format for all the three of them as:

`pragma(`[*declaration,...*]`)`
`entry(`*goal_pattern*, [*declaration,...*]`)`
`exit(`*goal_pattern*, [*declaration,...*]`)`

This uniform format allows the same syntax for the declarations of properties to be used both in `pragma/1` annotations and in `entry/2` and `exit/2` directives. Despite this, `pragma/1` is richer in its own nature, this meaning that some declarations at the goal level (thus appearing in a `pragma/1` annotation) cannot appear inside predicate level annotations, due to the lack of the appropriate context (e.g., because they refer to relationships concerning other goals variables — cf. sharing information).

We also propose the use of other declarative–like declarations. Most of them do not state any property of the program behaviour, but are directed to help the compiler to decide the safety/correctness of the program with respect to non–logical criteria (extra- or meta–logical, from some point of view) and even to change the semantics of the program, i.e., its run–time expected behaviour, although they also allow for capturing useful information:

- dynamic predicates: `dynamic/2`
- predicate suspension: `when/2`
- module system: `module/3`, `use/1`
- side–effects: `side_effect/2`
- meta–logical: `meta_logical/2`

4 Declarations of Properties: Capturing the Information

The declarations of properties that we propose are summarized in the following list and explained in the next sections.

- modes: `mode/2`, `data/2`
- types: `type/2`
- state of instantiation: `g/1`, `f/1`, `nf/1`, `ng/1`, `s/1`, `ns/1`, `term/2`, `linear/1`, `nonlinear/1`, `aliased/2`, `covers/2`
- dependencies: `dep/1`, `dep/2`, `indep/1`, `indep/2`, `share/1`
- granularity: `measure/2`, `domain/2`, `size/2`, `solutions/1`, `time/1`, `space/2`
- goal execution behaviour: `determinate/0`, `solutions/1`, `successful/0`
- program execution behaviour: `delayed/2`

Most of them follow a variable–oriented pattern allowing two arguments where the first is a variable to which the information in the second argument relates to. Where this pattern is used hereafter, and if nothing else is said, it should be assumed that a list of variables can appear also as first argument; the information then relates to all of them.

4.1 Declarations oriented to automatically–inferred information

The above declarations will appear inside `pragma/1` and `entry/2` or `exit/2` annotations. In doing this, we have followed an approach oriented towards a "global information domain." We consider an abstract global domain where information about the program can be stated in "sentences" (the declarations), thus allowing for communicating this information. For this communication a semantics of the sentences is needed. The formal semantics of each piece of information should be defined in terms of any other abstract domain possibly used for inferring information about the properties of the program.

This approach presents two advantages. On one hand, as we have said, it is easily extensible, provided one defines the semantics for the new properties one wants to add. On the other hand, it is also independent from any other abstract domain. Thus, any tool (or user) which (who) wants to communicate via this domain, might only be able to define semantic functions capturing the desired properties. In the following sections the intended semantics of the declarations is (informally) presented, which can be viewed as the semantics a user should give to them.

A different approach than that explained above could have been pursued. One could define the declarations for properties to be oriented to the different tools that may communicate among them. Thus, these declarations would encapsulate the particular information a particular tool is able to infer or understand. For example, imagine two different analysers over abstract domains `domain1` and `domain2`, and it has been decided to give the information encapsulated in terms with these names, the annotations would be as follows:

```
:- entry(goal_pattern,[domain1(...),domain2(...)]).

:- exit(goal_pattern,[domain1(...),domain2(...)]).
```

This approach is also easily extensible if more tools are to be communicated using this interface. But it is not so easily extensible if one can and wants to state other additional properties that the already existing tools might somehow take advantage of. It also has two other disadvantages. The so called semantic functions above mentioned have to be defined pairwise, i.e. one for each two different domains to be communicated. And, secondly, the user would have to be aware of several (possibly overlapping) domains of information, one for each different tool.

4.2 Combined information and combining declarations

As has already been said, we have followed a "global" approach to allow for ease of combination of distinct information in a general framework, thus defining a language for this purpose. The issue of the meaning of the combined information should be addressed via semantic functions. In this section we address the counterpart issue of the syntax for combining different assertions for the same predicate or goal.

In our approach, the information on properties of the program is given by means of two complementary constructions:

- different declarations for different properties, and
- a conjunction of such declarations (i.e. the lists of declarations in `pragma/1`, `entry/2` and `exit/2`).

In general, the information can be expressed as a logical formula in conjunctive or disjunctive normal form. This will allow for any combination of conjunctions and disjunctions of pieces of information, either about the same or different properties. A general conjunction is provided by the above mentioned lists; an inner disjunction in these ones can be provided by allowing for alternative declarations for the same property (as we will see with, for example, `term/2`). Inner disjunctions for different properties can be unfolded to outer disjunctions at the cost of repeating some information.

An outer disjunction can be expressed by having more than one `entry/2` or `exit/2` declaration per predicate. It is more or less intuitive that the meaning of these should be a disjunction, but a *closed* one. Thus, the properties specified in one of them may occur, but in any case, no other properties than those specified may occur (i.e. the list of all given declarations for a predicate is considered closed). The same is applicable to more than one `pragma/1` annotation.

An interesting matter is that of having the possibility of relating information at the entry and the exit points of a goal, either at the goal–level or at the predicate–level.

For this, a natural extension to the annotations proposed is to add to them an extra (optional) position which will act as a key. The extended annotations will look like:

```
pragma(key:[declaration,...])
entry(goal_pattern,key:[declaration,...])
exit(goal_pattern,key:[declaration,...])
```

where the key can be any constant, or — more classically — a number, acting as a reference relating distinct annotations at different (entry or exit) points. The same key appearing in two (or more) annotations at different points will state that whenever the properties specified at the entry point hold (and only when they hold), then those specified with the same key at the exit point will also hold.

Additionally, the key position could be extended to a list of keys to avoid the inconvenience of having to repeat pieces of information across disjunctions. Keys in annotations at the same point would then mean that whenever the properties stated in one of them hold, those of the other(s) will also hold, thus overlaying the (implicit) disjunction and turning it into a conjunction. Nonetheless, this could complicate annotations more than it facilitates the task of annotating, from our point of view, so it will not be included in this proposal.

5 Degree of Instantiation: Modes

Classical modes [Qui86] refer to the instantiation state of arguments at predicate entry and exit:

- `+` argument ground at predicate entry
- `-` argument ground at predicate exit

Standardized modes [PRO93] somewhat extend classical modes to encompass cases not covered by the latter:

- `+` argument not a variable at predicate entry
- `-` argument not a variable at predicate exit
- `@` argument not further instantiated from predicate entry to exit

Modes are a sort of abbreviated type declarations which refer only to the degree of instantiation required before and expected after the call to a predicate. The underlying idea below mode declarations is that some predicates are always called (or, better, should be called in order to work properly) with a given degree of instantiation for some variables. This associates a certain dataflow–oriented flavour with them, making an implicit distinction among *input* and *output* predicates. Thus, mode declarations are intended to express the expected instantiation degree at entry and exit of a predicate. To deal with this, and extend it to more refined and expressive annotations, we propose the following declaration:

`mode`(*var*, *degree*)

where the degree of instantiation is expressed by (more refined expressions will be allowed for this in different declarations in the next section):

`g`	ground variable
`f`	free variable
`nf`	non free variable
`ng`	non ground variable
`s`	non ground, non free variable, i.e. bound to a term with variables
`ns`	not bound to a term with variables, i.e. either ground or free

The mode declarations at entry and exit of different predicates inherited by the program variables must *unify* in the domain given by the mode declaration specs. This unification is, in general, a *one–way* unification, in order to reflect the execution flow of data. The different degree of instantiation for a variable between the `entry/2` and `exit/2` directives (or `pragma/1` annotations) will determine the character of input and output of such argument.

Thus, the classical approaches to modes can be modelled by the one herein presented, except for the "further instantiated" character of a variable. For this a declaration oriented to data–flow information might be available (see below).

Additionally, the mode specs can be used as functors of arity 1 to specify in a more concise way the instantiation degree of a variable (or more). Thus, the following declarations are also admitted:

`g`(*var*) `f`(*var*) `nf`(*var*) `ng`(*var*) `s`(*var*) `ns`(*var*)

5.1 Data–flow oriented modes

In addition to the "modes" view of the previous section, a kind of mode declaration which can be useful is a summarized form of the data–flow information that the different degrees of instantiation at entry and exit implies. For this, we propose to define the *mode* of an argument (or, better, a variable) as:

- **input**: if its value is required in the computation of the goal — a variable will *not* be input if it can be replaced by a *new* free variable without the computation being affected,
- **output**: if it is further instantiated upon execution of the goal

and propose the following data–flow modes:

- At entry point:
  - `+` if it is input
  - `-` if it is not input

- At exit point:
  - `+` if it is output
  - `-` if it is not output

Note that a variable can be both input and output: these modes, as defined, are not exclusive. The declaration `data/2` is defined to handle this kind of modes at the variable level as:

`data`(*var*, *data_mode*)

In order to embed mode information inside `pragma/1` annotations, the `data/2` declarations must be qualified. Thus:

```
:- ..., p(X,Y), pragma([data(-X,+),data(-Y,-),data(+X,+)]), p(Y,X), ...
```

means that `X` is output for the first `p/2` goal and also input for the second one, and `Y` is not output for the first one. It does not say anything else.

Declarations appearing in `pragma/1` must then be qualified so that they can be related to the goal to which they refer. The qualification consists of a `+` or `-` preceding the variable in `data/2`. The scope of a declaration with `-` is thus the first goal of the predicate involved that appears *before* the `pragma/1`, and for that with `+` the first goal appearing *after* the `pragma/1`.

6 Patterns of Instantiation: a Type System?

An extension to the above proposed modes which is closer to a type system — although it can also be viewed as a "depth–k" domain abstraction — is given by patterns of instantiation constructed from:

- a set of constants
- a set of functors
- a set of variables

which can be viewed as a natural extension of the above domain of "instantiation degree" to a richer domain. Patterns of instantiation will be used in a declaration of the form:

`term`(*var*, *term_degree*)

In particular, the sets mentioned above can be those of the program constant atoms, functors, and variables (but there is no special reason to make this restriction). Furthermore, in the case of variables, if the program variables are considered, then the language of these annotations allows for implicit declarations of properties. For example, an annotation containing:

`term(X,Y), term(Y,f(Z)), term(Z,g(W,W))`

also implies that `X` is aliased to `Y`, `Y` contains `Z` (and so does `X`), `Z` contains `W` (and so do `Y` and `X`) and also that all of them contain repeated variables. Thus, the information contained in the annotation is much more than what is directly explicit in it.

Such information goes beyond that on instantiation to also specifying aliasing and independence information, and complicates the interpretation of such declarations (although it facilitates stating such information). This can be avoided by forcing a normalization in which only new and distinct variables should appear in the second argument of a `term/2` declaration and providing for additional declarations for other kinds of information. We think that the idiom proposed (including program variables) is very rich, and worth the complication in its interpretation. Nonetheless, additional declarations are also provided for ease of use:

`linear`(*var*)
`nonlinear`(*var*)
`aliased`(*var*, *var*)
`covers`(*var*, *var*)

A variable is linear [Son86] if it is bound to a term containing only distinct variables (if any); it is not linear if at least one variable in the term to which it is bound to appears more than once. Two variables are aliased if they are bound to the same term (or to each other). One variable covers another(s) if it is bound to a term that contains it (them). Note the relations between these declarations and those of dependencies between variables (see next section 7).

Note that with the `term/2` declaration a disjunction of terms (i.e. a "possibility" that the variable may be instantiated to different terms) may be handled by having distinct declarations for the same variable.

A more refined type domain will need a well–defined type system. There is also a chance for including a type system for the language, if so desired, having a "(canonical) recursive (polymorphic) type" interpretation [MO84, Zob87, JB92] for:

`type`(*var*, *type*)

7 Variable Dependencies

The canonical declaration for expressing variable dependencies is:

`dep`([v_1, v_2, ..., v_n])

which expresses that the terms v_1 through v_n have a variable in common, i.e., further instantiation of one of them might lead to a further instantiation of all the others.

For denotational convenience, we introduce the following declarations:

- `indep([`v_1, v_2, ..., v_n`])` $\equiv \neg$ `dep([`v_1, v_2, ..., v_n`])`
- `dep(`v_1, v_2`)` $\equiv$ `dep([`v_1, v_2`])`
- `indep(`v_1, v_2`)` $\equiv$ `indep([`v_1, v_2`])`

Also the following equivalence of the `g/1` declaration holds:

- `g(`v`)` $\equiv$ `indep(`v, v`)`

Note that there is no direct equivalence between the `dep/1`, `indep/1` declarations and their pairwise counterparts. Thus, it is not true that `indep([X,Y,Z])`$\Rightarrow$ `indep(X,Y)`$\wedge$ `indep(X,Z)`$\wedge$`indep(Y,Z)`, although the opposite implication is true, as for example in {X$\rightarrow$A, Y$\rightarrow$f(A,B), Z$\rightarrow$B}. The case of `dep/1` and `dep/2` is just the opposite: `dep([X,Y,Z])`$\Rightarrow$`dep(X,Y)`$\wedge$`dep(X,Z)`$\wedge$`dep(Y,Z)` holds, whereas the opposite implication is not true.

As the other declarations presented, these ones have a connotation of definiteness, i.e. the dependencies that they state *will* occur. On the contrary, variable sharing information as inferred by many abstract interpretation tools [JL92, MH92] have a connotation of possibility: they inform of a dependency that *may* occur. In order to also handle this information, an additional declaration is proposed:

`share([[`t_1,t_2,...`],...])`

stating that any of the possible dependencies represented by the sharing sets might occur, but no dependency will occur which is not included in the given list[1].

8 Granularity of Goals

A useful task a compiler can perform is to derive the granularity of goals [DLH90]. For this task, it turns out that the availability of size measures of the arguments of the goals can be very helpful. In order for an analysis to infer these measures the program can be annotated with information regarding the unit of these measures, such as:

`measure(`*var*, *measure_name*`)`

where the legal measure names are (as in, for example, CasLog [DL93]):

[1] In the context of section 4.2 this can be viewed as an inner folding of a disjunction that could be expressed at the outer level of annotations.

`void`	irrelevant (goal granularity does not depend on this position)
`int`	integer–value
`length`	list–length
`size`	term–size
`depth([`*ChildList*`])`	term–depth, where *ChildList* is the list of recursive positions

In order to improve the estimation of relation size, the user can use additional declarations to declare the domain information for some predicates that range over finite domains. The declaration will be:

`domain`(*var*, *domain_name*)

where the legal domain names are (as in, for example, CasLog):

- an integer interval L-U, representing all the integers from the integer L to the integer U, or
- an enumerated set, i.e. a list of atoms, representing the elements in the domain.

Note that the domain expressions in a domain declaration must be of the same type. These expressions might be related to the type system used, if any, or, desirably, uniformly expressed within it. In the case that the system allows for constraint programming extensions within particular domains, it will also be desirable to have these constraint extensions be consistent with the domain declarations[2].

In order to also allow making the particular granularity of goals explicit (be it by the user him/herself or by a program analysis) other declarations may be useful:

`size`(*var*, *size_measure*)
`solutions`(*lower_bound-upper_bound*)
`time`(*time_measure*)

Traditionally, goal granularity has been expressed in terms of sizes of its arguments, number of solutions, and time complexity. The intended use of the above declarations is precisely this one: when related to a (most general) goal pattern of a given predicate the sizes of its arguments can be expressed with `size/2`. The declaration `solutions/1` gives lower and upper bounds to the number of solutions and `time/1` the time complexity (w.r.t. some metric) for a given state of instantiation of the goal, which will be expressed by making use of other declarations in the same annotation where these ones appear. We have added to the previous ones [Yan93] a space–based unit of measure specified with `space/2`. This unit can be very much implementation–dependent, so an extra argument has been added to the declaration in order to recognize the kind of metric which is used for the measure.

[2]In particular, in ElipSys [Eli92] the enumerated–set domain is restricted to integers.

Note that all the declarations referring to variables have to be qualified in the style of those for modes if placed within `pragma/1` annotations. The declarations `solutions/1` and `time/1` would refer, in these cases, to the goal to their left. Note also that the conditions upon which the properties stated hold are left to the rest of the annotations, in particular it is intended that the mode and instantiation state declarations are used for this purpose.

9 Determinacy of Goal Execution

This section introduces declarations for the programmer/high level compiler to express knowledge regarding the degree of determinacy of a given predicate or call. This can eventually help to produce better code. Determinism inference is intimately related to type and mode inference [DW89], since for some predicates it is impossible to make any assertion about its determinism without previous information about the type and mode of their arguments. Thus, the declarations herein proposed are strongly related to that of patterns and degree of instantiation, in that they establish the determinacy character of goals of a predicate in the case that certain such properties hold.

We distinguish the following classes of determinism. Two of them can be declared making use of `solutions/1`, for the other one a new declaration `determinate/0` is introduced. Again, the declarations can appear at goal level or at predicate level. If at goal level, they will then refer to the goal appearing to their left.

- Successful literal: it provides at least one solution: `solutions(1-n)`
- Determinate literal: if only one of its clauses will match: `determinate`
- Single solution literal: it provides at most one solution (*deterministic* in the classical sense): `solutions(0-1)`

The differences between a *single solution literal* and a *determinate literal* are that the first provides one solution, but it can match more than one clause, whereas determinate literals match only one clause. Note that the latter can still provide more than one solution through deeper backtracking.

It is interesting to extend the class of the determinate predicates to encompass those predicates whose clauses perform simple ("flat") tests concerning freeness, arithmetic comparison, etc. This aids program transformations such as automatic deallocation of shallow choice–points and delaying of noisy unifications.

In the spirit of already proposed rules for goal reduction selection (as the so called "Andorra Principle" [War88]) it will also be interesting to extend the class of determinate literals in a different direction: establishing an order among the goals based on the number of clauses each can reduce with. For this purpose, an extra argument can be added to `determinate/0` specifying this in the style of `solutions/1`.

10 Module System

A module system is a language feature of widely recognized help in program development [Mil86, O'K85b, GM86]. Modules provide for structuring, hiding, reuse of code, use of libraries and separate compilation, all of which facilitate the task of the program developer. On the other hand, modules make things more difficult to the compiler task of program optimization, in particular that of program global analysis, because they hide some parts of the program, which may not be accessible to the compiler as a unit. In contrast, they can be useful in providing a limited scope for extra–logical and side–effects predicates whose effects can complicate the analysis, such as assert.

In general, a module system must provide solutions for the following problems:

- identification of required and provided resources of each module,
- fulfillment of all required resources of all modules,
- avoidance of procedure name clashes between modules,
- compositionality of modules to form another module, and additionally
- reduction of interference with program optimization

Traditionally, the resources provided by each module, as well as its requirements, have been identified via declarations that explicitly point at them, the so called *exports* and *imports* of a module. In some approaches, the imports need not be declared, instead they are identified by the compiler (or even dynamically) from the module code. In our point of view explicit declarations not only allow viewing each module as a self–contained unit, but also contribute to make all available information explicit, in the spirit of this report, and because of these are very much recommended. For the same reasons we propose these declarations to be also *static*, in the sense of self–contained and closed, so that they can not be dynamically incremented. Thus, a module will be declared making use of:

```
:- module(module_name, [predicate_spec,...])
:- use([predicate_spec,...])
```

where the first one is mandatory and the second one, which refers to the imports list, optional. Using a declarative to appear in programs forces the assumption of a one–to–one correspondence between modules and files.

Once requirements and resources can be identified, they have to pair up in a cross–product fashion so that for every required resource of a module there is some other module providing it — thus, each module providing a predicate as a resource to other module has to be identified. Again, this can be done explicitly by the programmer,

or implicitly by the compiler. In the first approach, the user provides the names of the modules supplying the imports for any other module, either as a declaration or as a qualification of the procedure names themselves (the so called *module name expansion*). In the second approach, the compiler resolves the matching between imports and exports. Although getting rid of module names makes the system more flexible, names are sometimes needed for qualifying imported predicates that may clash. Also, identifying the suppliers of exported predicates for a module can be very helpful in modular program analysis. For these purposes, although our system does not need module name expansion[3], it allows giving a handle to the compiler to solve the "call graph" of a modular program by means of extending `use/1` as follows:

`use(`[*imported_module,...*]`)`

where *imported_module* stands for a term:

`module(`*module_name, file_name,* [*imported_predicate_spec,...*]`).`

In order to avoid making the system too rigid, a second form for `use/1` with and extra argument — `use(`*module_name*, [*imported_module,...*]`)` — can be used as a goal to override the existing declarative for some module (if called *before* compiling such a module — otherwise it will have no effect), which is identified by the module name in the first argument. Additionally it can be used to solve name clashes between imported predicates into a module: a special renaming construct is provided for the list of imported predicates — *predicate_spec* `->` *predicate_spec* — which means that the predicate used in the module, the first argument, corresponds in the supplier module to the exported predicate appearing as second argument. As for *private* predicates, i.e. those not exported, as opposed to those exported traditionally called *public*, name clashing must be automatically solved by the compiler.

The problem of coupling imports and exports via the modules that provide them, or solving the cross–references among modules, is related to the issue of compositionality of modules and the type of module system under consideration. In a *flat* module system all exports of a module are visible to every other module, no notion of compositionality is applied. In a *hierarchical* system modules are composed following a hierarchy explicited by the programmer. Other composition rules have been proposed (there are a number of compositional *algebras* for this purpose) but their discussion is outside the scope of this report. We only mention that in a compositional system the composition rules usually provide for identification of the modules which make predicates available as other module resources, whereas in flat systems this is not always the case. Because of its simplicity, we propose to use a flat module system, although providing for cross–referencing of modules via the `use/1` declarative.

[3] As all exports must be declared and all imports are static, solving references between modules is little more than matching module declarations. There is an exception for this: meta–calls between modules can not be solved at compile–time (see meta–predicates section 12).

Also the `use/1` declarative can help, as already mentioned, in modular global analysis of a modular program. This analysis can also be aided by using `entry/2` and `exit/2` declaratives for the exported and imported predicates. In particular, query–driven analysers require declaring the entry points for the program, which can be done using `module/3` (the entry points will be the exports of the main module), and additional `entry/2` declarations for these points will help the analysis. In the case of including in a module `exit/2` declarations for its imports, the assertion made above that the list of `exit/2` declarations for a given predicate is considered closed (section 4.2 on annotations) must be taken under the scope of the whole program, not each single module.

11 Dynamic Predicates

Predicates that are dynamically asserted/retracted at execution time must be declared explicitly with a declarative:

`dynamic`(*declared_type*, *predicate_spec*)

where the declared types of dynamic predicates can be:

`data`	if only facts are asserted
`memo`	if only logical consequences of the program itself are asserted
`call`	in other case

A `data` predicate[4] can be viewed as a term that could be recorded/retrieved instead: calls and retracts of it have both the same effect — data retrieval (no further computation is performed). In this case, calling the predicate is not harmful for any program analysis that may be performed, and thus it is allowed. This is also the case for `memo` predicates when performing abstract interpretation based analysis, although other kinds of analyses (as granularity analysis) can be complicated by these predicates.

In the case of a `call` predicate, any asserted/retracted instance of the predicate alters the semantics of the program. The problem here is that calls to such predicates lead to complex subcomputations which are not known at compile–time. No analysis can deal accurately with such programs. In order to allow the compilation process to keep the "undecidability" caused by the use of assert/retract (and abolish) under known bounds dynamic predicates may not be allowed to call non–dynamic predicates. To achieve this the `call` dynamic predicate type can be unfolded into:

`local_call`	if it only calls other dynamic predicates
`global_call`	if it can call any other predicate

[4] Clauses of a predicate which are asserted/retracted but never called can be considered as a set of facts for the term `:-/2` — thus such a predicate will fall under the `data` type.

For the same reasons `compile/1` should be avoided if any optimization based on program analysis is to be done. For similar reasons, too, only predicates declared dynamic should be used as dynamic, stressing the fact that asserting to a non–declared predicate does *not* make it become dynamic.

The classical `dynamic/1` is not supported. Or, on the other hand, it may be viewed as equivalent to `dynamic(call,` *predicate_spec*`)`.

12 Meta–Predicates

Predicates that use other predicates as arguments (meta–predicates) often can not be manipulated because it can not be known in advance which other predicates they are calling to (i.e. those they use as arguments). The problem is fixed around the meta–predicate built–ins (call/1, bagof/3, findall/3, setof/3, \+/1) but can be generalized to user meta–predicates.

The solution to the problem posed by meta–predicates relies on knowing the range of variables in meta–calls[5]. Thus an annotation like:

```
:- ..., pragma([term(X,p(Y))]), call(X), ...
```

will allow global analysis to proceed.

Such a knowledge can be inferred by a types/state of instantiation analysis, or, otherwise, made explicit by the user. Care should be taken in the analyses to couple with complex meta–calls:

```
:- ..., pragma([term(X,';'(p(Y),q(Z)))]), call(X), ...
```

should be understood as one of the following constructions, that can be achieved by simple program transformation:

```
:- ..., foo(Y,Z), ...
```

```
foo(X,Y):- p(X).
foo(X,Y):- q(Y).
```

```
:- ..., call(p(Y);q(Z)), ...
```

The predicates used as arguments of other predicates may belong to distinct modules, in which case they should have been declared in the corresponding declarations for modules (see modules section 10). Nonetheless, if the calls are not known until execution, and no module–name qualification is allowed, the system must be able to resolve the module–name expansion at run–time.

[5]The same problem occurs with I/O builtins, and a similar solution is applicable.

The influence of unknown meta–calls can be bounded to keep it under control by a simple program transformation:

```
:- goal.                                    :- '$query_goal'.

goal:- ...,                                 '$query_goal':- ...,
       call(X),                                             call(X),
       ...                                                  ...
```

All the original program, including the query goals, is renamed apart. Thus, the renamed program can be analysed, despite the meta–calls, until the points where these occur. At these points, the original program will be entered, and this will occur at execution–time, at any of its predicates (as the meta–call goal is not known), and control will finally return to the renamed program. Analysis of the original program would probably lead to unuseful information — too general to be practically used — but this is not propagated to the renamed program, that could be optimized in some cases. Thus, analysis can proceed with the renamed program assuming that no information is available at the points of meta–calls, and avoiding having to analyse the original program, which in any case might not be optimized.

Such a transformation will lead to duplication of the amount of code, where half this code would possibly be optimized, but not the other half. The user may be aware of this and have the opportunity to switch this off if he/she so desires.

13 Goal Suspension

Most systems declare goals that can be suspended at the concrete point where the particular goal is called. We argue that it is more natural to declare them at the predicate level, allowing for the required flexibility to distinguish between different types of calls to the same predicate and different conditions for them. This is achieved by allowing an extensible format (in the style of `pragma/1`) in the declaration:

`when(`*goal_pattern*`, [`*condition*`,...])`

The conditions upon goal patterns can then be specified with any of the (suitable) declarations that `pragma/1` accepts. These include of course tests on the instantiation degree of goal arguments, i.e. the most usual conditions for suspension which are already present in existing languages, including the classical `nonvar/1`, `ground/1`, and so on. They also include conditions in the style of the declarations for determinacy of goals, including, at least, `determinate/0`, which we think is a useful extension to embed the style of coroutining present in Andorra–like languages. Additionally, the list of conditions allows for further extension, and thus it is possible to include, if desired, more elaborated conditions, as for example the SICStus2.1 `?=/2`, or conditions on variable dependencies, as long as the underlying system supports them. Such conditions may

or may not be understood by the analysis part of the compiler, thus possibly assuming the implied information at the point of awakening of goals for their optimization.

Note that the list of conditions in the `when/2` declaration is understood as a *conjunction* of them, whereas *disjunctions* of conditions will be handled by having separate declarations for the same predicate.

Also, different conditions for different goals in the program for the same predicate can be handled with a simple renaming of goals. This allows for the goal–level style of suspensions of some languages. Thus,

```
goal:- ..., when(ground(X),p(X,Y)), ..., when(indep(Y,W),p(Y,W)), ...
```

can be expressed at the predicate–level by:

```
:- when(p1(X,_),[g(X)]).
:- when(p2(X,Y),[indep(X,Y)]).

goal:- ..., p1(X,Y), ..., p2(Y,W), ...
```

It would be very desirable to have information available at compile–time on the scope of suspensions in the program. This can be determined by means of a declaration to occur in `pragma/1` annotations (or at predicate–level `entry/2` or `exit/2` declaratives), such as:

`delayed(`[*goal_pattern,...*]`,` [*condition,...*]`)`

with which a list of goals (specified by their patterns) which will be suspended (at the specified conditions) can be declared.

14 Issues related to Parallelism

There exist many proposals to exploit parallelism in logic programming, a number of which focus in Prolog in particular and attempt to make this exploitation guaranteeing preservation of Prolog semantics [DeG87, CC89, HG90, AK90, SCWY90, Eli92]. In this objective, all of them have to deal with several problems posed mainly by the non–logical features of Prolog, specially side–effects and pruning. Another issue which has to be taken into account when optimizing programs with for example reordering, as well as parallelism, is the meta–logical nature of some built–in predicates. These issues are the topic of this section.

14.1 Side–effect predicates

In general, in an execution of a parallel Prolog, side–effects can not be allowed to execute freely in parallel with other goals. A mechanism of synchronization must be

provided in order to prevent a side–effect from being executed before other preceding (in the sense of the sequential operational semantics) side–effects or goals, in the cases when such adherence to the sequential order is desired, i.e. if a behaviour of the program identical to that observable on a sequential Prolog implementation is to be preserved.

In order to preserve the sequential observable behaviour, side–effects can only be executed when every subgoal to their left has been executed, i.e. when they are "leftmost" in the execution tree. However, a distinction can be made between *soft* and *hard* side–effects (a side–effect is regarded to be *hard* if it could affect subsequent execution, see [DeG87] and [MH89]). This distinction allows more parallelism. It is also convenient in this context to distinguish between side–effect builtins and side–effect procedures, i.e. those procedures that have side–effects in their clauses or call other side–effect procedures. This information can be made available to a compiler and/or produced through global analysis by means of a declaration:

`side_effect`(*predicate_spec*, type)

where the side–effect type can be either `soft` or `hard` and the goal type `builtin` or `procedure`.

Knowledge on side–effect builtins is inherent to the compiler, whereas that on procedures can be easily achieved by a simple analysis that propagates "upwards" the side–effect nature of builtins appearing in the program.

To achieve side–effect synchronization, various compile–time methods are possible, depending on the kind(s) of parallelism a particular system exploits. Nonetheless, most systems, being and- or or–parallel or both, define "parallel side–effects" for each sequential counterpart: p_write, p_read, p_assert, ... These ones do not preserve the sequential semantics of the original ones, and can be used when this preservation is not desired.

14.2 Meta–logical predicates

As well as the side–effects, there are a number of built–in predicates, which can be regarded as "impure" (in the sense of variable binding sensitive), which are worth annotating when doing program manipulation and optimization. These predicates (the most relevant being `var/1`) can cause incorrect propagation of bindings during partial evaluation[6] or incorrect reordering optimizations.

To take these predicates into account, we propose a declaration `meta_logical/2` for "purity" of predicates:

`meta_logical`(*pred_spec*, *type*)

[6]For example, given `(var(X), X=0)`, a partial evaluator might unfold it to var(0), which has a different operational semantics

where the argument *type* can be either `impure` or `pure`, specifying a predicate to be variable binding sensitive or not, respectively.

14.3 Pruning predicates

Pruning in Prolog, and specifically the cut control rule, is inherently sequential. Thus, to preserve it in a parallel implementation some synchronization solution is needed. As in the case of side–effects, also alternative structures have been proposed, the most relevant of it being the so called *commit* operator. Also a more natural construction than the cut has been proposed: "quiet" if–then–elses [O'K85a].

14.4 Catching exceptions: error handling

Besides pruning, there is the problem of handling exceptions. In summary, there exist two alternatives to facilitate error handling and solve the associated problems:

- to restrict the scope of error catching,
- to enlarge its scope,

or, alternatively, to drop the exception–handling catch/throw procedures. The usual exception–handling mechanism (included in the Prolog standard) makes use of the predicate:

`catch(`*goal*`,`*event*`,`*handler*`)`

where any exception matching *event* raised (by `throw/1`) during execution of *goal* will be caught by the mentioned *handler*. Yet another alternative is to replace the `catch/3` with:

`catch(`*event*, *handler*`)`

with the following meaning: "I do know how to recover from exception *event*. The predicate to be evaluated for such recovering is *handler*". Thus, `catch/2` is equivalent to `catch/3` but instead of specifying the goal where the exception is caught, it is effective for the remaining and–or tree.

The idea is to treat catch/throw as if they were plain goals. So the catch is visible to the siblings of the current goal. This approach allows the events to be treated at the abstraction level they should be. Two separate modules could be written to be fault tolerant if the exception scheme is clearly defined. No modification of a module which already catches exceptions is needed to achieve full exception handling in the whole system.

15 User–level Syntactic Sugar: Macros

This section introduces some *macros* that can be translated into the declarations presented above, which the compiler can understand. The purpose of these macros is to relieve the programmer from using such low–level declarations and still allow him/her to define and make some knowledge about the program that is too high–level to be used directly by the compilers explicit.

One such macro can be a declaration for functions. A function will be defined as a procedure which, for given *input* arguments, returns unique values for the *output* arguments. The function macro must provide means to specify the input/output relation as well as, arguably, the pattern of the input arguments which achieves functionality. The syntax will therefore resemble that of mode and degree of instantiation of arguments, but allowing for a more compact format:

```
:- function(p(g,-)).
```

will be translated to:

```
:- entry(p(X,Y),[g(X),data(X,+),data(Y,-),solutions(1)]).
:- exit(p(X,Y),[g([X,Y])]).
```

The declarative `function/1` will then have as argument a pattern of the goal, or a list of patterns, specifying the required instantiation degree for the input arguments (as in `mode/2`) and which arguments are output (as in `data/2`).

16 Conclusion: Towards a Global Abstract Domain

As we said in section 4.2 we have followed an approach oriented towards a global domain where every piece of information one wants to assert over a program can be asserted. We would like to consider an abstract global domain where information about the program can be stated in sentences with a formal semantics. The language for these sentences is that of first order logic. We have made an attempt at a first definition towards a domain like that, allowing for capturing the information and combining them through conjunctions and disjunctions.

A formal treatment of the information domain should also be pursued. The formal semantics of each piece of information should be defined in terms of any other abstract domain possibly used for inferring information about the properties of the program. The intuition behind this is that abstract interpreters infer properties about execution states of the program which are referred to terms to which variables are bound, relationships between variables, etc. — the properties. The sentences about these properties can be viewed as descriptions of abstractions corresponding to particular items of the concrete

domain. They are themselves abstractions, other than those of the particular abstract domains used for specific analysis. The language of these sentences will then be viewed as an abstraction of the abstractions themselves.

Different abstract interpreters could then be interfaced via this language. It is not possible to do this by concretizing the sentences and then abstracting to the particular abstract domain, since concretization is not always finitely computable. A different approach is to consider semantic functions defined for each different abstract domain: these will give the semantics of the statements in each particular domain. Thus, the declarations used to make such statements will have the same semantics of the abstraction functions for the different abstract domains. These can be viewed as (supra-) concretization functions for the global domain over each (already abstract) domain. The counterpart (supra-) abstraction functions should also be defined.

As an example, consider the declaration for groundness `g/1`. We can give it an informal semantics as follows: `g(X)` $\equiv$ "variable `X` is bound to a ground term." A formal semantics could be defined as: `g(X)` $\equiv$ `var_abstraction(freeness,{X},ground)`, in a particular domain of "freeness." The function

`var_abstraction(`*AbstractDomain, Variables, AbstractValue*`)`

defines the abstraction in *AbstractDomain* of the terms to which *Variable* can be bounded in the concrete domain for which *AbstractValue* is the abstraction in the global domain.

To be able to relate or combine information for different abstract domains in order to obtain more precise information, relationships between the different properties should be formally defined. Consider the following example, where an abstract interpreter has proved that:

```
entry(p(X,Y,Z),1:[s(X),g(Y)]),
exit(p(X,Y,Z),1:[g(Z)]).
```

and a different one that:

```
entry(p(X,Y,Z),2:[term(X,p(A)),f(A),g(Y),share([[X,Z]])]),
exit(p(X,Y,Z),2:[share([[X,Z]])]).
```

Having the knowledge about how to relate different abstraction functions, the following will be derived:

$$term(X, p(A)), f(A) \Rightarrow\ s(X)$$
$$s(X), g(Y), share([[X, Z]]) \Rightarrow\ share([[X, Z]]), g(Z)$$
$$share([[X, Z]]), g(Z) \Rightarrow\ g(X), share([])$$

thus, we augment the last annotation with this result to give:

```
entry(p(X,Y,Z),2:[term(X,p(A)),f(A),g(Y),share([[X,Z]])]),
exit(p(X,Y,Z),2:[g(X),share([])]).
```

The conclusion (and our proposal for future work) is that having an abstract global unification function, one can perform in the global abstract domain the usual operations of unification and propagation. This will achieve more precise information, in addition to a framework were abstract interpreters (and other tools) can be combined. In fact, the global domain can be viewed as a combination of many others: those related to properties herein defined, on one hand, plus those particular to the interpreters to be interfaced, on the other. The semantic functions to concretize and abstract from such a global domain to the particular ones could then be handled by already existing approaches to combining abstract domains.

References

[AK90] K.A.M. Ali and R. Karlsson. Full Prolog and Scheduling Or-parallelism in Muse. *International Journal of Parallel Programming*, 1990. Vol. 19, No. 6, pp. 445–475.

[Car88] M. Carlsson. *Sicstus Prolog User's Manual.* Po Box 1263, S-16313 Spanga, Sweden, February 1988.

[CC89] S.-E. Chang and Y. P. Chiang. Restricted AND-Parallelism Execution Model with Side-Effects. In Ewing L. Lusk and Ross A. Overbeek, editors, *Proceedings of the North American Conference on Logic Programming*, pages 350–368. MIT Press, Cambridge, MA, 1989.

[DeG87] D. DeGroot. Restricted AND-Parallelism and Side-Effects. In *International Symposium on Logic Programming*, pages 80–89. San Francisco, IEEE Computer Society, August 1987.

[DL93] S.K. Debray and N.W. Lin. Cost analysis of logic programs. *ACM Transactions on Programming Languages and Systems*, 15(5):826–875, 1993.

[DLH90] S. K. Debray, N.-W. Lin, and M. Hermenegildo. Task Granularity Analysis in Logic Programs. In *Proc. of the 1990 ACM Conf. on Programming Language Design and Implementation*, pages 174–188. ACM Press, June 1990.

[DW89] S. K. Debray and D. S. Warren. Functional Computations in Logic Programs. *ACM Transactions on Programming Languages and Systems*, 11(3):451–481, 1989.

[Eli92] *Elipsys User Manual—Release Version 0.5*, November 1992.

[GM86] J.A. Goguen and J. Meseguer. Eqlog: equality, types, and generic modules for logic programming. In *Logic Programming: Functions, Relations, and Equations*, Englewood Cliffs, 1986. Prentice-Hall.

[HG90] M. Hermenegildo and K. Greene. &-Prolog and its Performance: Exploiting Independent And-Parallelism. In *1990 International Conference on Logic Programming*, pages 253–268. MIT Press, June 1990.

[JB92] G. Janssens and M. Bruynooghe. Deriving Descriptions of Possible Values of Program Variables by means of Abstract Interpretation. *Journal of Logic Programming*, 13(2 and 3):205–258, July 1992.

[JL92] D. Jacobs and A. Langen. Static Analysis of Logic Programs for Independent And-Parallelism. *Journal of Logic Programming*, 13(2 and 3):291–314, July 1992.

[MH89] K. Muthukumar and M. Hermenegildo. Complete and Efficient Methods for Supporting Side Effects in Independent/Restricted And-parallelism. In *1989 International Conference on Logic Programming*, pages 80–101. MIT Press, June 1989.

[MH92] K. Muthukumar and M. Hermenegildo. Compile-time Derivation of Variable Dependency Using Abstract Interpretation. *Journal of Logic Programming*, 13(2 and 3):315–347, July 1992.

[Mil86] D.A. Miller. A theory of modules for logic programming. In *IEEE Symposium on Logic Programming*, pages 106–114, Salt Lake City, Utah, September 1986. IEEE Computer Society.

[MO84] A. Mycroft and R.A. O'Keefe. A Polymorphic Type System for Prolog. *Artificial Intelligence*, 23:295–307, 1984.

[oBLdM92] University of Bristol, Katholieke Universiteit Leuven, and Universidad Politécnica de Madrid. Interface between the prince prolog analysers and the compiler. Technical Report KUL/PRINCE/92.1, Katholieke Universiteit Leuven, October 1992.

[O'K85a] R. A. O'Keefe. On the treatment of cuts in Prolog source-level tools. In *Symposium on Logic Programming*, pages 68–72. IEEE Computer Society, Technical Committee on Computer Languages, The Computer Society Press, July 1985.

[O'K85b] R.A. O'Keefe. Towards an algebra for constructing logic programs. In *IEEE Symposium on Logic Programming*, pages 152–160, Boston, Massachusetts, July 1985. IEEE Computer Society.

[PRO93] International Organization for Standardization, National Physical Laboratory, Teddington, Middlesex, England. *PROLOG*, 1993.

[Qui86] *Quintus Prolog User's Guide and Reference Manual—Version 6*, April 1986.

[SCWY90] V. Santos-Costa, D.H.D. Warren, and R. Yang. Andorra-I: A Parallel Prolog System that Transparently Exploits both And- and Or-parallelism. In *Proceedings of the 3rd. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.* ACM, April 1990.

[Son86] H. Sondergaard. An application of abstract interpretation of logic programs: occur check reduction. In *European Symposium on Programming, LNCS 123*, pages 327–338. Springer-Verlag, 1986.

[War88] D.H.D. Warren. The Andorra Model. Presented at Gigalips Project workshop. U. of Manchester, March 1988.

[Yan93] Shan-When Yan. A general-purpose abstract interpreter and its application to cost analysis. *ParForce* esprit project report d.wp.1.3.1.m1.1, CEC, July 1993.

[Zob87] J. Zobel. Derivation of Polymorphic Types for Prolog Programs. In *Fourth International Conference on Logic Programming*, pages 817–838. University of Melbourne, MIT Press, May 1987.