A New Module System for Prolog*

Daniel Cabeza and Manuel Hermenegildo

Department of Computer Science, Technical U. of Madrid (UPM)
dcabeza@fi.upm.es, herme@fi.upm.es

Abstract. It is now widely accepted that separating programs into modules is useful in program development and maintenance. While many Prolog implementations include useful module systems, we argue that these systems can be improved in a number of ways, such as, for example, being more amenable to effective global analysis and transformation and allowing separate compilation or sensible creation of standalone executables. We discuss a number of issues related to the design of such an improved module system for Prolog and propose some novel solutions. Based on this, we present the choices made in the Ciao module system, which has been designed to meet a number of objectives: allowing separate compilation, extensibility in features and in syntax, amenability to modular global analysis and transformation, enhanced error detection, support for meta-programming and higher-order, compatibility to the extent possible with official and de-facto standards, etc.
Keywords: Modules, Modular Program Processing, Global Analysis and Transformation, Separate Compilation, Prolog, Ciao-Prolog.

1 Introduction

Modularity is a basic notion in modern computer languages. Modules allow dividing programs into several parts, which have their own independent name spaces and a clear interface with the rest of the program. Experience has shown that there are at least two important advantages to such program modularization. The first one is that being able to look at parts of a program in a more or less isolated way allows a divide-and-conquer approach to program development and maintenance. For example, it allows a programmer to develop or update a module at a time or several programmers to work on different modules in parallel. The second advantage is in efficiency: tools which process programs can be more efficient if they can work on a single module at a time. For example, after a change to a program module the compiler needs to recompile only that module (and perhaps a few related modules). Another example is a program

* This work was supported in part by the "EDIPIA" (CICYT TIC99-1151) and "ECCOSIC" (Fulbright 98059) projects. The authors would like to thank Francisco Bueno and the anonymous referees for their useful comments on previous versions of this document. The Ciao system is a collaborative international effort and includes contributions from members of several institutions, which are too many to mention here: a complete list can be found in the Ciao system documentation.

verifier which is applied to one module at a time and does its job assuming some properties of other modules. Also, modularity is also one of the fundamental principles behind object-oriented programming.

The topic of modules and logic programming has received considerable attention (see, for example, [23, 9, 34, 13, 21, 22]). Currently, many popular Prolog systems such as Quintus [28] and SICStus [8] include module systems which have proved very useful in practice.[1] However, these practical module systems also have a series of shortcomings, specially with respect to effectively supporting separate program compilation, debugging, and optimization.

Our objective is to discuss from a practical point of view a number of issues related to the design of an improved module system for Prolog and, based on this, to present the choices made in the module system of Ciao Prolog [2].[2] Ciao Prolog is a next-generation logic programming system which, among other features, has been designed with modular incremental compilation, global analysis, debugging, and specialization in mind. The module system has been designed to stay as similar as possible to the module systems of the most popular Prolog implementations and the ISO-Prolog module standard currently being finished [20], but with a number of crucial changes that achieve the previously mentioned design objectives. We believe that it would not be difficult to incorporate these changes in the ISO-Prolog module standard or in other module systems. The rest of the paper proceeds as follows: Section 2 discusses the objectives of the desired module system and Section 3 discusses some of the issues involved in meeting these objectives. Section 4 then describes the Ciao Prolog module system. Within this section, Subsection 4.5 discusses some enhancements to standard Prolog syntax extension facilities. Finally, Section 5 describes the notion of packages, a flexible mechanism for implementing modular language extensions and restrictions, which emerges naturally from the module system design. An example of a package is provided which illustrates some of the advantages of this design. Because of space restrictions and because the focus is on the motivations behind the choices made, the presentation is informal.

2 Objectives in the Design of the Ciao Module System

We start by stating the main objectives that we have had in mind during the design of the Ciao module system:

- *Allowing modular (separate) and efficient compilation.* This means that it should be possible to compile (or, in general, process) a module without having to compile the code of the related modules. This allows for example having pre-compiled (pre-processed, in general) system or user-defined libraries. It also allows the incremental and parallel development of large software projects.

[1] Surprisingly, though, it is also true that a number of Prolog systems do not have any module system at all.

[2] The Ciao system can be downloaded from `http://www.clip.dia.fi.upm.es/Software`.

- *Local extensibility/restriction, in features and in syntax.* This means that it should be possible to define syntactic and semantic extensions and restrictions of the language in a local way, i.e., so that they affect only selected modules. This is very important in the context of Ciao, since one of its objectives is to serve as an experimental workbench for new extensions to logic programming (provided that they can be translated to the core language).
- *Amenability to modular global analysis.* We foresee a much larger role for global analysis of logic programs, not only in the more traditional application of optimization [35, 33, 31, 4], but also in new applications related to program development, such as automated debugging, validation, and program transformation [3, 10, 5, 16, 17]. This is specially important in Ciao because the program development environment already includes a global analysis and transformation tool (`ciaopp`, the Ciao preprocessor [17, 15]) which performs these tasks and which in our experience to date has shown to be an invaluable help in program development and maintenance.
- *Amenability to error detection.* This means that it should be possible to check statically the interfaces between the modules and detect errors such as undefined predicates, incompatible arities and types, etc.
- *Support for meta-programming and higher-order.* This means that it should be possible to do meta- and higher-order programming across modules without too much burden on the programmer. Also, in combination with the previous point, it should be possible to detect errors (such as calls to undefined predicates) on sufficiently determined higher-order calls.
- *Compatibility with official and de-facto standards.* To the extent possible (i.e., without giving up other major objectives to fulfill this one) the module system should be compatible with those of popular Prolog systems (e.g., Quintus/SICStus) and official standards, such as the core ISO-Prolog standard [19, 12] and the current drafts of the ISO-Prolog module standards [20]. This is because it is also a design objective of Ciao that it be (thanks to a particular set of libraries which is loaded by default) a standard Prolog system. This is in contrast to systems like Mercury [30] or Goedel [18] which are more radical departures from Prolog. This means that the module system will be (at least by default) *predicate-based* rather than *atom-based* (as in XSB [29] and BIM [32]), i.e., it will provide separation of predicate symbols, but not of atom names. Also, the module system should not require the language to become strongly typed, since traditional Prologs are untyped.[3]

3 Discussion of the Main Issues Involved

None of the module systems used by current Prolog implementations fulfill all of the above stated objectives, and some include characteristics which are in clear

[3] Note however, that this does not prevent having voluntary type declarations or more general assertions, as is indeed done in Ciao [25, 26].

opposition to such objectives.[4] Thus, we set out to develop an improved design. We start by discussing a number of desirable characteristics of the module system in order to fulfill our objectives. Amenability to global analysis and being able to deal with the core ISO-Prolog standard features were discussed at length in [3], where many novel solutions to the problems involved were proposed. However, the emphasis of that paper was not on modular analysis. Herein, we will choose from some of the solutions proposed in [3] and provide further solutions for the issues that are more specific to modular analysis and to separate compilation.[5]

- *Syntax, flags, etc. should be local to modules.* The syntax or mode of compilation of a module should not be modified by unrelated modules, since otherwise separate compilation and modular analysis would be impossible. Also, it should be possible to use different syntactic extensions (such as operator declarations or term expansions) in different modules without them interacting. I.e., it should be possible to use the same operator in different modules with different precedences and meanings. In most current module systems for Prolog this does not hold because syntactic extensions and compilation parameters (e.g., Prolog flags) are global. As a result, a module can be compiled in radically different ways depending on the operators, expansions, Prolog flags, etc. set by previously loaded modules or simply typed into the top level. Also, using a syntactic extension in a module prevents the use of, e.g., the involved operators in other modules in a different way, making the development of optional language extensions very complicated. In conclusion, we feel that directives such as `op/3` and `set_prolog_flag/2` must be local to a module.
- *The entry points of a module should be statically defined.* Thus, the only external calls allowed from other modules should be to exported predicates. Note that modules contain code which is usually related in some way to that of other modules. A good design for a modular program should produce a set of modules such that each module can be understood independently of the rest of the program and such that the communication (dependencies) among the different modules is as reduced as possible. By a *strict* module system we refer to one in which a module can only communicate with other modules via its *interface* (this interface usually contains data such as the names of the *exported* predicates). Other modules can only use predicates which are among the ones exported by the considered module. Predicates which are not exported are not visible outside the module. Many current module systems for Prolog are not strict and allow calling a procedure of a module even if it is not exported by the module. This clearly defeats the purpose of the module system and, in addition, has a catastrophic impact

[4] Unfortunately, lack of space prevents us from making detailed comparisons with other individual module systems. Instead, we discuss throughout the paper advantages and disadvantages of particular solutions present in different current designs.

[5] We concentrate here on the design on the module system. The issue of how this module system is applied to modular analysis is addressed in more detail in [27].

on the precision of global analysis, precluding many program optimizations. Thus, we feel that the module system should be strict.

- *Module qualification is for disambiguating predicate names, not for changing naming context.* This a requirement of separate compilation (processing) since otherwise to compile (process) a module it may be necessary to know the imports/exports of all other modules. As an example, given a call `m:p` ("call `p` in module `m`"), with the proposed semantics the compiler only needs to know the exports of module `m`. If qualification meant changing naming context, since module `m` can import predicate `p` from another module, and that module from another, the interfaces of all those modules would have to be read. Furthermore, in some situations changing naming context could invalidate the strictness of the module system.
- *Module text should not be in unavailable or unrelated parts.* This means that all parts of a module should be within the module itself or directly accessible at the time of compilation, i.e., the compiler must be able to automatically and independently access the complete source of the module being processed.[6]
- *Dynamic parts should be isolated as much as possible.* Dynamic code modification, such as arbitrary runtime clause addition (by the use of assert-like predicates), while very useful in some applications, has the disadvantage that it adds new entry points to predicates which are not "visible" at compile-time and are thus very detrimental to global analysis [3]. One first idea is to relegate such predicates to a library module, which has to be loaded explicitly.[7] In that way, only the modules using those functionalities have to be specially handled, and the fact that such predicates are used can be determined statically. Also, in our experience, dynamic predicates are very often used only to implement "global variables", and for this purpose a facility for adding facts to the program suffices. This simpler feature, provided that this kind of dynamic predicates are declared as such explicitly in the source, pose no big problems to modular global analysis. To this end, Ciao provides a set of builtins for adding and deleting facts to a special class of dynamic predicates, called "data predicates" (`asserta_fact/1`, `retract_fact/1`, etc), which are declared as "`:- data ...`" (similar kinds of dynamic predicates are mentioned in [11]). Furthermore, the implementation of such data predicates can be made much more efficient than that of the normal dynamic predicates, due to their restricted nature.
- *Most "built-ins" should be in libraries which can be loaded and/or unloaded from the context of a given module.* This is a requirement related to extensibility and also to more specific needs, such as those of the previous point, where it was argued that program modification "built-ins" should be

[6] Note that this is not the case with the classical *user files* used in non-modular Prolog systems: code used by a user file may be in a different user file with no explicit relation with the first one (there is no usage declaration that allows relating them).

[7] Note, however, that in Ciao, to preserve compatibility for older programs, a special case is implemented: if no library modules are explicitly loaded, then all the modules containing the ISO predicates are loaded by default.

relegated to a library. The idea is to have a core language with very few predefined predicates (if any) and which should be a (hopefully pure) subset of ISO-Prolog. This makes it possible to develop alternative languages defining, for example, alternative I/O predicates, and to use them in a given module while others perhaps use full ISO-Prolog. It also makes it easier to produce small executables.

– *Directives should not be queries.* Traditionally, directives (clauses starting with "`:-`") were executed by the Prolog interpreter as queries. While this makes some sense in an interpretative environment, where program compilation, load (linking), and startup are simultaneous, is does not in other environments (and, specially, in the context of separate compilation) in which program compilation, linking, and startup occur at separate times. For example, some of the directives used traditionally are meant as instructions for the compiler while, e.g., others are used as initialization goals. Fortunately, this is well clarified in the current ISO standard [19, 12], where declarations are clearly separated from initialization goals.
– *Meta-predicates should be declared, at least if they are exported, and the declaration must reflect the type of meta-information handled in each argument.* This is needed in order to be able to perform a reasonable amount of error checking for meta-predicates and also to be able to statically resolve meta-calls across modules in most cases.

4 The Ciao Module System

Given the premises of previous sections, we now proceed to present their concretization in the Ciao module system.

4.1 General Issues

Defining Modules: The source of a Ciao module is typically contained in a single file, whose name must be the same as the name of the module, except that it may have an optional `.pl` extension. Nevertheless, the system allows inclusion of source from another file at a precise point in the module, by using the ISO-Prolog [19, 12] `:- include` declaration. In any case, such included files must be present at the time of processing the module and can for all purposes be considered as an integral part of the module text. The fact that the file contains a module (as opposed to, e.g., being a user file –see below) is flagged by the presence of a "`:- module(...`" declaration at the beginning of the file.

For the reasons mentioned in Section 2 the Ciao module system is, as in most logic programming system implementations, predicate-based (but only by default, see below). This means that non-exported predicate names are local to a module, but all functor and atom names in data are shared. We have found that this choice does provide the needed capabilities most of the time, without imposing too much burden on the user or on the implementation. The advantage of this, other than compatibility, and probably the reason why this option has been chosen traditionally, is that it is more concise for typical Prolog programs in which many atoms and functors are shared (and would thus have to be exported in an atom-based system). On the other hand, it forces having to deal specially with meta-programming, since in that case functors can become predicate names and vice-versa. It can also complicate having truly abstract data types in modules. The meta-predicate problem is solved in Ciao through suitable declarations (see Section 4.4). Also, in order to allow defining truly abstract data types in Ciao, it is possible to *hide* atom/functor names, i.e., make them local to a module, by means of "`:- hide ...`" declarations, which provide an automatic renaming of such symbols. This does not prevent a program from creating data *of that type* if meta-predicates such as "`=..`" are loaded and used, but it does prevent creating and matching such data using unification. Thus, in contrast to predicate names, which are local unless explicitly exported, functor and atom names are exported by default unless a `:- hide` declaration is used.[8]

Imports, Exports, and Reexports: A number of predicates in the module can be *exported*, i.e., made available outside the module, via explicit `:- export` declarations or in an export list in the `:- module(...` declaration. It is also possible to state that *all* predicates in the module are exported (by using '_').

It is possible to *import* a number of individual predicates or also all predicates from another module, by using `:- use_module` declarations. In any case it is only possible to import from a module predicates that it exports. It is possible to import a predicate which has the same name/arity as a local predicate. It is also possible to import several predicates with the same name from different modules. This applies also to predicates belonging to *implicitly-imported modules*, which play the role of the built-ins in other logic programming systems. In Ciao there are really no "built-ins": all system predicates are (at least conceptually) defined in libraries which have to be loaded for these predicates to be accessible to the module. However, for compatibility with ISO, a set of these libraries implementing the standard set of ISO builtins is loaded by default.

A module `m1` can *reexport* another module, `m2`, via a `:- reexport` declaration. The effect of this is that `m1` exports all predicates of `m2` as if they had been defined in `m1` in the same way as they are defined in `m2`. This allows implementing modules which *extend* other modules (or, in object-oriented terms, classes which inherit from other classes [24]). It is also possible to reexport only some of the predicates of another module, by providing an explicit list in the `:- reexport` declaration, *restricting* that module.

In Ciao it is possible to mark certain predicates as being *properties*. Examples of properties are *regular types*, *pure properties* (such as `sorted`), *instantiation properties* (such as `var`, `indep`, or `ground`), *computational properties* (such as `det` or `fails`), etc. Such properties, since they are actually predicates, can be exported or imported using the same rules as any other predicate. Imported properties can be used in assertions (declarations stating certain characteristics

[8] This feature of being able to hide functor and atom names is not implemented in the distribution version of Ciao as of the time of writing of this paper (Vers. 1.4).

of the program, such as, e.g., preconditions and postconditions) in the same way as locally defined ones. This allows defining, e.g., the abstract data types mentioned above. This is discussed in more detail in the descriptions of the Ciao assertion language [2, 25] and the Ciao preprocessor [17, 15].

Visibility Rules: The predicates which are *visible* in a module are the predicates defined in that module plus the predicates imported from other modules. It is possible to refer to predicates with or without a *module qualification.* A module-qualified predicate name has the form *module:predicate* as in the call `lists:append(A,B,C)`. We call *default module* for a given predicate name the module which contains the definition of the predicate which will be called when using the predicate name without module qualification, i.e., when calling `append(A,B,C)` instead of `lists:append(A,B,C)`. Module qualification makes it possible to refer to a predicate from a module which is not the default for that predicate name.

We now state the rules used to determine the default module of a given predicate name. If the predicate is defined in the module in which the call occurs, then this module is the default module. I.e., local definitions have priority over imported definitions. Otherwise, the default module is the *last module* from which the predicate is imported in the module text. Also, predicates which are explicitly imported (i.e. listed in the importation list of a `:- use_module`) have priority over those which are imported implicitly (i.e. imported when importing all predicates of a module). As implicitly-imported modules are considered to be imported first, the system allows the redefinition of "builtins". By combining implicit and explicit calls it is also possible not only to redefine builtins, but also to *extend* them, a feature often used in the implementation of many Ciao libraries. Overall, the rules are designed so that it is possible to have a similar form of inheritance to that found in object-oriented programming languages (in Ciao this also allows supporting a class/object system naturally as a simple extension of the module system [24]). It is not possible to access predicates which are not imported from a module, even if module qualification is used and even if the module exports them. It is also not possible to define clauses of predicates belonging to other modules, except if the predicate is defined as dynamic and exported by the module in which it is defined.

Additional rules govern the case when a module redefines predicates that it also reexports, which allows making specialized modules which are the same as a reexported module but with some of the predicates redefined as determined by local predicate definitions (i.e., *instances* of a module/class, in object-oriented terms –see the Ciao manual [2] for details).

4.2 User Files and Multifile Predicates

For reasons mainly of backwards compatibility with non-modular Prolog systems, there are some deviations from the visibility rules above which are common to other modular logic programming systems [28, 8]: the "`user`" module and *multifile* predicates.

User Files: To provide backwards compatibility with non-modular code, all code belonging to files which have no module declaration is assumed to belong to a single special module called "`user`". These files are called "user files", as opposed to calling them modules (or packages –see later). All predicates in the user module are "exported". It is possible to make unrestricted calls from any predicate defined in a user file to any other predicate defined in another user file. However, and differently to other Prolog systems, predicates imported from a normal module into a user file are not visible in the other user files unless they are explicitly imported there as well. This at least allows performing separate static compilation of each user file, as all static predicate calls in a file are defined by reading only that file. Predicates defined in user files can be visible in regular modules, but such modules must explicitly import the "user" module, stating explicitly which predicates are imported from it.

The use of user files is discouraged because, apart from losing the separation of predicate names, their structure makes it impossible to detect many errors that the compiler detects in modules by looking at the module itself (and perhaps the interfaces of related modules). As an example, consider detecting undefined predicates: this is not possible in user files because a missing predicate in a user file may be defined in another user file and used without explicitly importing it. Thus, it is only possible to detect a missing predicate by examining all user files of a project, which is itself typically an unknown (and, in fact, not even in this way, since that predicate could even be meant to be typed in at the top level after loading the user files!). Also, global analysis of user files typically involves considerable loss of precision because all predicates are possible entry points [3]. Note that it is often just as easy and flexible to use modules which export all predicates in place of user files (by simply adding a `:- module(_,_).` header to the file), while being able to retain many of the advantages of modules.

Multifile Predicates: Multifile predicates are a useful feature (also defined in ISO-Prolog) which allows a predicate to be defined by clauses belonging to different files (modules in the case of Ciao). To fit this in with the module system, in Ciao these predicates are implemented as if belonging to a special module `multifile`. However, calls present in a clause of a multifile predicate are always to visible predicates of the module where that clause resides. As a result, multifile predicates do not pose special problems to the global analyzer (which considers them exported predicates) nor to code processing in general.

4.3 Dynamic Modules

The module system described so far is quite flexible but it is *static*, i.e., except in user files, it is possible to determine statically the set of imports and exports of a given module and the set of related modules, and it is possible to statically resolve to which module each call in the program refers to. This has many advantages: modular programs can be implemented with no run-time overhead with respect to a non-modular system and it is also possible to perform extensive static analysis for optimization and error detection. However, in practice it is sometimes very useful to be able to load code dynamically and call it. In Ciao this is fully supported, but only if the special library `dynmods` which defines the appropriate builtins (e.g., `use_module`) is explicitly loaded (`dynmods` actually reexports a number of predicates from the compiler, itself another library). This can then be seen by compile-time tools which can act more conservatively if needed. Also, the adverse effects are limited to the module which imports the compiler.

4.4 Dealing with Meta-Calls

As mentioned before, the fact that the Ciao module system is predicate-based forces having to deal specially with meta-programming, since in that case functors can become predicate names and vice-versa. This problem is solved in Ciao, as in similar systems [28, 8] through `meta_predicate` declarations which specify which arguments of predicates contain meta-data. However, because of the richer set of higher-order facilities and predicate types provided by Ciao [6], there is a correspondingly richer set of types of meta-data (this also allows more error detection):

`goal`: denotes a goal (either a simple or a complex one) which will be called.
`clause`: denotes a clause, of a dynamic predicate, which will be asserted/retracted.
`fact`: denotes a fact (a head-only clause), of a `data` predicate.
`spec`: denotes a predicate name, given as *Functor/Arity* term (this kind of meta-term is used somewhat frequently in builtin predicates, but seldom in user-defined predicates).
`pred(N)`: denotes a predicate construct to be called by means of a `call/N` predicate call. That is, it should be an atom equal to the name of a predicate of arity N, a structure with functor the name of a predicate of arity M (greater than N) and with M-N arguments, or a predicate abstraction with N arguments.[9]
`addmodule`:
this in fact is not a real meta-data specification. Rather, it is used to pass, along with the predicate arguments, the calling module, to allow handling more involved meta-data (e.g., lists of goals) by using conversion builtins.[10]

The compiler, by knowing which predicates have meta-arguments, can verify if there are undetermined meta-calls (which for example affect the processing when performing global analysis), or else can determine (or approximate) the calls that these meta-arguments will produce.

4.5 Modular Syntax Enhancements

Traditionally (and also now in the ISO standard [19, 12]) Prolog systems have included the possibility of changing the syntax of the source code by the use

[9] A full explanation of this type of meta-term is outside the scope of this paper. See [6] for details.

[10] This a "low-level" solution, which can be a reasonable overall solution for systems without a type system. The higher-level solution in Ciao involves the combination of the type and meta-data declarations (currently in progress).

of the `op/3` builtin/directive. Furthermore, in many Prolog systems it is also possible to define *expansions* of the source code (essentially, a very rich form of "macros") by allowing the user to define (or extend) a predicate typically called `term_expansion/2` [28, 8]. This is usually how, e.g., definite clause grammars (DCG's) are implemented.

However, these features, in their original form, pose many problems for modular compilation or even for creating sensible standalone executables. First, the definitions of the operators and expansions are global, affecting a number of files. Furthermore, which files are affected cannot be determined statically, because these features are implemented as a side-effect, rather than a declaration, and they are meant to be active after they are read by the code processor (top-level, compiler, etc.) and remain active from then on. As a result, it is impossible by looking at a source code file to know if it will be affected by expansions or definitions of operators, which may completely change what the compiler really sees. Furthermore, these definitions also affect how a compiled program will read terms (when using the term I/O predicates), which will also be affected by operators and expansions. However, in practice it is often desirable to use a set of operators and expansions in the compilation process (which are typically related to source language enhancements) and a completely different set for reading or writing data (which can be related to data formatting or the definition of some application-specific language that the compiled program is processing). Finally, when creating executables, if the compile-time and run-time roles of expansions are not separated, then the code that defines the expansions must be included in the executable, even if it was only meant for use during compilation.

To solve these problems, in Ciao we have redesigned these features so that it is still possible to define source translations and operators but they are local to the module or user file defining them. Also, we have implemented these features in a way that has a well defined behavior in the context of a stand-alone compiler (the Ciao compiler, `ciaoc` [7]). In particular, the directive `load_compilation_module/1` allows separating code that will be used at compilation time from code which will be used at run-time. It loads the module defined by its argument *into the compiler* (if it has not been already loaded). It differs from the `use_module/1` declaration in that the latter defines a use by the module being compiled, but does not load the code into the compiler itself. This distinction also holds in the Ciao interactive top-level, in which the compiler (which is the same library used by `ciaoc`) is also a separate module.

In addition, in order to make the task of writing expansions easier,[11] the effects usually achieved through `term_expansion/2` can be obtained in Ciao by means of four different, more specialized directives, which, again, *affect only the current module*. Each one defines a different target for the translations, the first being equivalent to the `term_expansion/2` predicate which is most commonly included in Prolog implementations. The argument for all of them is a predicate indicator of arity 2 or 3. When reading a file, the compiler (actually, the general

[11] Note that, nevertheless, writing interesting and powerful translations is not necessarily a trivial task.

purpose module processing library –see [7]) invokes these translation predicates at the appropriate times, instantiating their first argument with the item to be translated (whose type varies from one kind of predicate to the other). If the predicate is of arity 3, the optional third argument is also instantiated with the name of the module where the translation is being done, which is sometimes needed during certain expansions. If the call to the expansion predicate is successful, the term returned by the predicate in the second argument is used to replace the original. Else, the original item is kept. The directives are:

`add_sentence_trans/1` : Declares a translation of the terms read by the compiler which affects the rest of the current text (module or user file). For each subsequent term (directive, fact, clause, ...) read by the compiler, the translation predicate is called to obtain a new term which will be used by the compiler in place of the term present in the file. An example of this kind of translation is that of DCG's.

`add_term_trans/1` : Declares a translation of the terms and sub-terms read by the compiler which affects the rest of the current text. This translation is performed after all translations defined by add_sentence_trans/1 are done. For each subsequent term read by the compiler, and recursively any subterm included in such a term, the translation predicate is called to possibly obtain a new term to replace the old one. Note that this is computationally intensive, but otherwise very useful to define translations which should affect any term read. For example, it is used to define *records* (feature terms [1]), in the Ciao standard library `argnames` (see 5.1).

`add_goal_trans/1` : Declares a translation of the goals present in the clauses of the current text. This translation is performed after all translations defined by `add_sentence_trans/1` and `add_term_trans/1` are done. For each clause read by the compiler, the translation predicate is called with each goal present in the clause to possibly obtain another goal to replace the original one, and the translation is subsequently applied to the resulting goal. Note that this process is aware of meta_predicate definitions. In the Ciao system, this feature is used for example in the `functions` library which provides functional syntax, as functions inside a goal add new goals before that one.

`add_clause_trans/1` : Declares a translation of the clauses of the current text. The translation is performed before `add_goal_trans/1` translations but after `add_sentence_trans/1` and `add_term_trans/1` translations. This kind of translation is defined for more involved translations and is related to the compiling procedure of Ciao. The usefulness of this translation is that information on the interface of related modules is available when it is performed, but on the other hand it must maintain the predicate defined by each clause, since the compiler has already made assumptions regarding which predicates are defined in the code. For example, the object-oriented extension of Ciao (O'Ciao) uses this feature [24].

Figure 1 shows, for an example clause of a program, to which subterms each type of translation would be applied, and also the order of translations. The

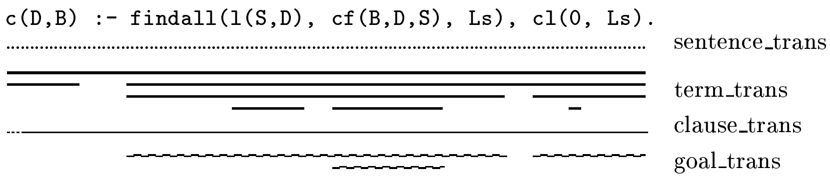

Fig. 1. Subterms to which each translation type is applied in a clause

principal functor of the head in the clause translation is dashed because the translation cannot change it.

Finally, there is another directive in Ciao related to syntax extension, whose raison d'être is the parametric and extensible nature of the compiler framework: `new_declaration/1` (there is also a `/2` variant). Note that in ISO-Standard Prolog declarations cannot be arbitrary Prolog goals. Thus, the Ciao compiler flags an error if a declaration is found which is not in a predefined set. A declaration `new_declaration(Decl)` can be used to declare that `Decl` is a valid declaration *in the rest of the current text* (module or user file). Such declarations are simply ignored by the compiler or top level, but can be used by other code processing programs. For example, in the Ciao system, program assertions and machine-readable comments are defined as new declarations and are processed by the `ciaopp` preprocessor and the `lpdoc` [14] automatic documenter.

5 Packages

Experience using the Ciao module system shows that the local nature of syntax extensions and the distinction between compile-time and run-time work results in the libraries defining extensions to the language having a well defined and repetitive structure. These libraries typically consist of a main source file which defines only some declarations (operator declarations, declarations loading other modules into the compiler or the module using the extension, etc.). This file is meant to be *included* as part of the file using the library, since, because of their local effect, such directives must be part of the code of the module which uses the library. Thus, we will call it the "include file". Any auxiliary code needed at compile-time (e.g., translations) is included in a separate module which is to be loaded into the compiler via a `load_compilation_module` directive which is placed in the include file. Also, any auxiliary code to be used at run-time is placed in another module, and the corresponding `use_module` declaration is also placed in the include file. Note that while this run-time code could also be inserted in the include file itself, it would then be replicated in each module that uses the library. Putting it in a module allows the code to be shared by all modules using the library.

Libraries constructed in this manner are called "packages" in Ciao. The main file of such a library is a file which is to be *included* in the importing module. Many libraries in Ciao are packages: `dcg` (definite clause grammars), `functions` (functional syntax), `class` (object oriented extension), `persdb` (persistent database), `assertions` (to include assertions –see [25, 26]), etc. Such libraries can be loaded using a declaration such as `:- include(library(functions))`. For convenience (and other reasons related to ISO compatibility), this can also be written as `:- use_package(functions)`.[12]

There is another feature which allows defining modules which do not start with a `:- module` declaration, and which is useful when defining language extensions: when the first declaration of a file is unknown, the declared library paths are browsed to find a package with the same name as the declaration, and if it is found the declaration is treated as a module declaration plus a declaration to use that package. For example, the package which implements the object oriented capabilities in Ciao is called "`class`": this way, one can start a class (a special module in Ciao) with the declaration "`:- class(myclass)`", which is then equivalent to defining a module which loads the `class` package. The `class` package then defines translations which transform the module code so that it can be used as a class, rather than as a simple module.

5.1 An Example Package: `argnames`

To clarify some of the concepts introduced in the paper, we will describe as an example the implementation of the Ciao library package "`argnames`".[13] This library implements a syntax to access term arguments by name (also known as *records*). For example, Fig. 2 shows a fragment of the famous "zebra" puzzle written using the package. The declaration `:- argnames` (where `argnames` is defined as an operator with suitable priority) assigns a name to each of the arguments of the functor `house/5`. From then on, it is possible to write a term with this functor by writing its name (`house`), then the infix operator `'$'`, and then, between brackets (which are as in ISO-Prolog), the arguments one wants to specify, using the infix operator `'=>'` between the name and the value. For example, `house${}` is equivalent in that code to `house(_,_,_,_,_)` and `house${nation=>Owns_zebra,pet=>zebra}` to `house(_,Owns_zebra,zebra,_,_)`.

The library which implements this feature is composed of two files, one which is the package itself, called `argnames`, and an auxiliary module which implements the code translations required, called `argnames_trans` (in this case no run-time code is necessary). They are shown in Appendix A (the transformation has been simplified for brevity by omitting error checking code).

The contents of package `argnames` are self-explanatory: first, it directs the compiler to load the module `argnames_trans` (if not already done before), which contains the code to make the required translations. Then, it declares a sentence translation, which will handle the argnames declarations, and a term translation,

[12] We are also considering adding a feature to allow loading packages using normal `:- use_module` declarations, which saves the user from having to determine whether what is being loaded is a package or an ordinary module.

[13] This package uses only a small part of the functionality described. Space restrictions do not allow adding a longer example or more examples. However, many such examples can be found in the Ciao system libraries.

```
:- use_package([argnames]).
:- argnames house(color, nation, pet, drink, car).
zebra(Owns_zebra, Drinks_water, Street) :-
    Street = [house${},house${},house${},house${},house${}],
    member(house${nation=>Owns_zebra,pet=>zebra}, Street),
    member(house${nation=>Drinks_water,drink=>water}, Street),
    member(house${drink=>coffee,color=>green}, Street),
    left_right(house${color=>ivory}, house${color=>green}, Street),
    member(house${car=>porsche,pet=>snails}, Street),
    ...
```

Fig. 2. "zebra" program using `argnames`

which will translate any terms written using the argnames syntax. Finally, it declares the operators used in the syntax. Recall that a module using this package is in fact including these declarations into its code, so the declarations are local to the module and will not affect the compilation of other modules.

The auxiliary module `argnames_trans` is also quite straightforward: it exports the two predicates which the compiler will use to do the translations. Then, it declares a data predicate (recall that this is a simplified dynamic predicate) which will store the declarations made in each module. Predicate `argnames_def/3` is simple: if the clause term is an `argnames` declaration, it translates it to nothing but stores its data in the above mentioned data predicate. Note that the third argument is instantiated by the compiler to the module where the translation is being made, and thus is used so that the declarations of a module are not mixed with the declarations in other modules. The second clause is executed when the end of the module is reached. It takes care of deleting the data pertaining to the current module. Then, predicate `argnames_use/3` is in charge of making the translation of argname'd-terms, using the data collected by the other predicate. Although more involved, it is a simple Prolog exercise.

Note that the argnames library only affects the modules that load it. Thus, the operators involved (`argnames`, `$`, `=>`) can be used in other modules or libraries for different purposes. This would be very difficult to do with the traditional model.

6 Conclusions

We have presented a new module system for Prolog which achieves a number of fundamental design objectives such as being more amenable to effective global analysis and translation, allowing separate compilation and sensible creation of standalone executables, extensibility/restriction in features and in syntax, etc. We have also shown in other work that this module system can be implemented easily [7] and can be applied successfully in several modular program processing tasks, from compilation to debugging to automatic documentation generation [7, 27, 17, 14]. The proposed module system has been designed to stay as similar as possible to the module systems of the most popular Prolog implementations and the ISO-Prolog module standard currently being finished, but with a number of crucial changes that achieve the previously mentioned design objectives. We believe that it would not be difficult to incorporate these changes in the ISO-Prolog module standard or in other module systems. In the latter case, the cost would be some minor backward-incompatibility with some of the existing modular code, but which could generally be fixed easily with a little rewriting. We argue that the advantages that we have pointed out clearly outweigh this inconvenience.

References

1. H. Aït-Kaci, A. Podelski, and G. Smolka. A feature-based constraint system for logic programming with entailment. In *Proc. Fifth Generation Computer Systems 1992*, pages 1012–1021, 1992.
2. F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López-García, and G. Puebla. The Ciao Prolog System. Reference Manual. TR CLIP3/97.1, School of Computer Science, Technical University of Madrid (UPM), August 1997.
3. F. Bueno, D. Cabeza, M. Hermenegildo, and G. Puebla. Global Analysis of Standard Prolog Programs. In *European Symposium on Programming*, number 1058 in LNCS, pages 108–124, Sweden, April 1996. Springer-Verlag.
4. F. Bueno, M. García de la Banda, and M. Hermenegildo. Effectiveness of Abstract Interpretation in Automatic Parallelization: A Case Study in Logic Programming. *ACM Trans. on Programming Languages and Systems*, 21(2):189–238, March 1999.
5. F. Bueno, P. Deransart, W. Drabent, G. Ferrand, M. Hermenegildo, J. Maluszynski, and G. Puebla. On the Role of Semantic Approximations in Validation and Diagnosis of Constraint Logic Programs. In *Int'l WS on Automated Debugging–AADEBUG'97*, pages 155–170, Sweden, May 1997. U. of Linköping Press.
6. D. Cabeza and M. Hermenegildo. Higher-order Logic Programming in Ciao. TR CLIP7/99.0, Facultad de Informática, UPM, September 1999.
7. D. Cabeza and M. Hermenegildo. The Ciao Modular Compiler and Its Generic Program Processing Library. In *ICLP'99 WS on Parallelism and Implementation of (C)LP Systems*, pages 147–164. N.M. State U., December 1999.
8. M. Carlsson and J. Widen. *Sicstus Prolog User's Manual*. Po Box 1263, S-16313 Spanga, Sweden, April 1994.
9. W. Chen. A theory of modules based on second-order logic. In *Proc. 4th IEEE Internat. Symposium on Logic Programming*, pages 24–33, San Francisco, 1987.
10. M. Comini, G. Levi, M. C. Meo, and G. Vitiello. Proving properties of logic programs by abstract diagnosis. In M. Dams, editor, *Analysis and Verification of Multiple-Agent Languages, 5th LOMAPS Workshop*, number 1192 in Lecture Notes in Computer Science, pages 22–50. Springer-Verlag, 1996.
11. S.K. Debray. Flow analysis of dynamic logic programs. *Journal of Logic Programming*, 7(2):149–176, September 1989.
12. P. Deransart, A. Ed-Dbali, and L. Cervoni. *Prolog: The Standard*. Springer, 1996.
13. 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.
14. M. Hermenegildo. A Documentation Generator for (C)LP Systems. In this volume: Proceedings of CL2000, LNCS, Springer-Verlag.
15. M. Hermenegildo, F. Bueno, G. Puebla, and P. López-García. Program Analysis, Debugging and Optimization Using the Ciao System Preprocessor. In *Proc. of ICLP'99*, pages 52–66, Cambridge, MA, November 1999. MIT Press.

16. M. Hermenegildo and The CLIP Group. Programming with Global Analysis. In *Proc. of ILPS'97*, pages 49–52, October 1997. MIT Press. (Invited talk abstract).
17. M. Hermenegildo, G. Puebla, and F. Bueno. Using Global Analysis, Partial Specifications, and an Extensible Assertion Language for Program Validation and Debugging. In K. R. Apt, V. Marek, M. Truszczynski, and D. S. Warren, editors, *The Logic Programming Paradigm: a 25–Year Perspective*, pages 161–192. Springer-Verlag, July 1999.
18. P. Hill and J. Lloyd. *The Goedel Programming Language*. MIT Press, 1994.
19. International Organization for Standardization. *PROLOG. ISO/IEC DIS 13211 — Part 1: General Core*, 1994.
20. International Organization for Standardization. *PROLOG. Working Draft 7.0 X3J17/95/1 — Part 2: Modules*, 1995.
21. D. Miller. A logical analysis of modules in logic programming. *Journal of Logic Programming*, pages 79–108, 1989.
22. L. Monteiro and A. Porto. Contextual logic programming. In *Proc. of ICLP'89*, pages 284–299. MIT Press, Cambridge, MA, 1989.
23. 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.
24. A. Pineda and M. Hermenegildo. O'ciao: An Object Oriented Programming Model for (Ciao) Prolog. TR CLIP 5/99.0, Facultad de Informática, UPM, July 1999.
25. G. Puebla, F. Bueno, and M. Hermenegildo. An Assertion Language for Debugging of Constraint Logic Programs. In *ILPS'97 WS on Tools and Environments for (C)LP*, October 1997. Available as TR CLIP2/97.1 from `ftp://clip.dia.fi.upm.es/pub/papers/assert_lang_tr_discipldeliv.ps.gz`.
26. G. Puebla, F. Bueno, and M. Hermenegildo. An Assertion Language for Debugging of Constraint Logic Programs. In *Analysis and Visualization Tools for Constraint Programming*, LNCS. Springer-Verlag, 2000. To appear.
27. G. Puebla and M. Hermenegildo. Some Issues in Analysis and Specialization of Modular Ciao-Prolog Programs. In *ICLP'99 WS on Optimization and Implementation of Declarative Languages*, pages 45–61. U. of Southampton, U.K, Nov. 1999.
28. *Quintus Prolog User's Guide and Reference Manual—Version 6*, April 1986.
29. K. Sagonas, T. Swift, and D.S. Warren. The XSB Programming System. In *ILPS WS on Programming with Logic Databases*, TR #1183, pages 164–164. U. of Wisconsin, October 1993.
30. Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury: an efficient purely declarative LP language. *JLP*, 29(1–3), October 1996.
31. A. Taylor. High performance prolog implementation through global analysis. Slides of the invited talk at PDK'91, Kaiserslautern, 1991.
32. P. Van Roy, B. Demoen, and Y. D. Willems. Improving the Execution Speed of Compiled Prolog with Modes, Clause Selection, and Determinism. In *Proceedings of TAPSOFT '87*, LNCS. Springer-Verlag, March 1987.
33. P. Van Roy and A.M. Despain. High-Performace Logic Programming with the Aquarius Prolog Compiler. *IEEE Computer Magazine*, pages 54–68, January 1992.
34. D.S. Warren and W. Chen. Formal semantics of a theory of modules. TR 87/11, SUNY at Stony Brook, 1987.
35. R. Warren, M. Hermenegildo, and S. K. Debray. On the Practicality of Global Flow Analysis of Logic Programs. In *Fifth International Conference and Symposium on Logic Programming*, pages 684–699. MIT Press, August 1988.

CLIP WS papers, TRs and manuals available at `http://www.clip.dia.fi.upm.es`

A Code for the Package argnames

The package `argnames`:

```
:- load_compilation_module(library(argnames_trans)).
:- add_sentence_trans(argnames_def/3).
:- add_term_trans(argnames_use/3).
:- op(150, xfx, [$]).
:- op(950, xfx, (=>)).
:- op(1150, fx, [argnames]).
```

The translation module `argnames_trans`:

```
:- module(argnames_trans, [argnames_def/3, argnames_use/3]).
:- data argnames/4.

argnames_def((:- argnames(R)), [], M) :-
        functor(R, F, N),
        assertz_fact(argnames(F,N,R,M)).
argnames_def(end_of_file, end_of_file, M) :-
        retractall_fact(argnames(_,_,_,M)).

argnames_use($(F,TheArgs), T, M) :-
        atom(F),
        argnames_args(TheArgs, Args),
        argnames_trans(F, Args, M, T).

argnames_args({}, []).
argnames_args({Args}, Args).

argnames_trans(F, Args, M, T) :-
        argnames(F, A, R, M),
        functor(T, F, A),
        insert_args(Args, R, A, T).

insert_args([], _, _, _).
insert_args('=>'(F,A), R, N, T) :-
        insert_arg(N, F, A, R, T).
insert_args(('=>'(F,A), As), R, N, T) :-
        insert_arg(N, F, A, R, T),
        insert_args(As, R, N, T).

insert_arg(N, F, A, R, T) :-
        N > 0,
        (   arg(N, R, F)
        ->  arg(N, T, A)
        ;   N1 is N-1,
            insert_arg(N1, F, A, R, T) ).
```