The Ciao Modular, Standalone Compiler and Its Generic Program Processing Library

Daniel Cabeza and *Manuel Hermenegildo*

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

Abstract: Ciao Prolog incorporates a module system which allows separate compilation and sensible creation of standalone executables. We describe some of the main aspects of the Ciao modular compiler, `ciaoc`, which takes advantage of the characteristics of the Ciao Prolog module system to automatically perform separate and incremental compilation and efficiently build small, standalone executables with competitive run-time performance. `ciaoc` can also detect statically a larger number of programming errors. We also present a generic code processing library for handling modular programs, which provides an important part of the functionality of `ciaoc`. This library allows the development of program analysis and transformation tools in a way that is to some extent orthogonal to the details of module system design, and has been used in the implementation of `ciaoc` and other Ciao system tools. We also describe the different types of executables which can be generated by the Ciao compiler, which offer different tradeoffs between executable size, startup time, and portability, depending, among other factors, on the linking regime used (static, dynamic, lazy, etc.). Finally, we provide experimental data which illustrate these tradeoffs.
Keywords: Prolog, Separate Compilation, Program Processing, Executable types, Modules, Ciao-Prolog.

1. Introduction

Ciao Prolog [CLI97] 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 Ciao module system [CH99a], while attempting to be compatible to a large extent with official and de-facto standards (i.e., with popular Prolog implementations and the ISO-Prolog module standard [PRO95]), includes a number of crucial changes that arguably enable building a better language and program development environment. In this paper we describe the overall architecture of the Ciao standalone compiler, `ciaoc`, which takes advantage of the characteristics of the module system to achieve a number of global design objectives, including detecting a larger number of errors statically, performing modular incremental ("separate") compilation, supporting modular extensibility of the language in features and in syntax, efficiently building small, standalone executables with different executable size tradeoffs and competitive run-time performance, offering support for meta-programming and higher-order, and allowing global analysis, debugging, and specialization/optimization.

`ciaoc` does not treat the compilation process as a translation from a single, isolated Prolog source to, e.g., its WAM bytecode. Instead, a module is compiled taking into account its relationship with the modules it uses. Also, sets of modules comprising an application, together with the set of user or system libraries it uses, are processed globally and incrementally. As a result, the corresponding object code and any other processing output is kept always updated with respect to the source while recompiling the minimal required set of dependent modules after a change. This applies also when the compiler is used in the toplevel shell, which also tracks in the same way which loaded modules have changed and need updating. A correct image of the program is always kept in the toplevel without predicate duplications or old code lingering around, as can possibly happen with traditional toplevel shells. `ciaoc` treats modularly and incrementally other information in addition to module interfaces, such as the annotated and optimized programs produced by the `ciaopp` preprocessor [HBPL99].

One of the most interesting results of the development of the compiler, from the software architecture point of view, has been that we have been able to abstract away into a generic code processing library much of the functionality related to the modular and incremental treatment of programs, even if they are multi-file and use multiple user and system libraries. This library allows the development of program analysis and transformation tools in a way that is to some extent orthogonal to the details of the module system design, and has been used also in other Ciao system tools such as the `ciaopp` preprocessor, the automatic documenter [Her99], etc. In fact, the whole compiler is itself also a library, which can be used by any Ciao executable. This has made it very easy for example to have a standalone command-line compiler (as is usual in other programming languages) and a script interpreter, along with the more typical Prolog interactive shell, and to ensure that they all behave in exactly the same way.

The rest of the paper proceeds as follows: Section 2. presents the generic code processing library and its interaction with the Ciao module system. Section 3. briefly describes the compiler itself, `ciaoc`. Section 4. describes a number of interesting errors that `ciaoc` can detect statically. Section 5. presents the different types of executables that `ciaoc` can build. Section 6. presents performance data, which illustrate the different tradeoffs involved. Section 7. compares the `ciaoc` compiler with some other compilers. Finally, Section 8. presents our conclusions.

The complete Ciao system, including the compiler and library described in the paper, as well as other related software can be downloaded from the CLIP Software WWW site:

`http://www.clip.dia.upm.es/Software`

2. The Ciao Generic Code-processing Framework

In our experience with code processing tools in general and with global analysis tools in particular (such as the PLAI analyzer [MH92, BdlBH99]), we have came across the difficulty of being able to reproduce exactly the logic of the compiler when reading source files. Note that in a practical system it is necessary to deal with syntax extensions (operators, expansions, etc.), inclusions of code, redefinition of prolog flags, modules, imports, exports, local definitions, reexports, visibility rules, external code, etc. We have come to the conclusion that the best way of reproducing exactly the compiler processing logic is to factor out this part into a library module such that the same code is used for all code processing tools. However, because these tools perform different jobs the library must be appropriately parametrized. To this end, we have implemented in Ciao a generic code processing framework which is used by the (WAM) compiler and also by the rest of our code processing tools. Currently, this framework is used by, in addition to the low-level compiler, all the preprocessor components (global analyzers, parallelizers, specializers, etc.), the automatic documentation generator, and the assertion processing library [CLI97]. Note that, for simplicity, in the following we sometimes refer to this code processing framework as "the compiler", and to process a module with it to "compile" such module.

2.1. Modular processing

Within the framework, modules are processed in a separate fashion, that is, the "processing" (compilation, analysis, etc) is applied to one module at a time, but in such a way that the necessary information from related modules is available. This information on a module, which other modules using it need when being processed, is usually called the *interface* of the module. In other modular languages, such as Modula, in which there is a *definition* part and an *implementation* part for each module, such information is stored in the definition part. In the Ciao module system [CH99a], as in that of other Prolog systems (e.g., [Swe95, Qui86]), the definition part is included in the source along with the implementation part, for programmer convenience. However, in order to obtain and store the interface separately and fully support separate compilation, in Ciao the compiler automatically extracts the interface definition from the source file. The advantage of this approach is that the user does not need to write a separate definition part as this is automatically done by the compiler. The extracted definition is stored by the compiler in a separate file from the one containing the actual code of the module: the *interface* file. From that point on, the interface file will be used by the compiler any time the interface part of the corresponding module is needed. For brevity, we will call such files "`.itf`" files, in reference to the ending used in the interface files managed by `ciaoc`. However, note that different tools may need additional data from related modules. For example, several Ciao tools need information on the assertions present in the program, and this is stored in additional `.asr` files.

The process of extracting the interface information is performed the first time a module is processed, either by direct request or as a result of the processing of another module which uses it. At any time, the `.itf` files are automatically managed by the framework, which regenerates them when the source of the associated module changes. Processing a module requires only its code and the interface files of the modules imported by that module (i.e., the source of these modules is not necessary). The visibility of the `.pl` and `.itf` files can be controlled with standard file access permissions.[1]

One advantage of this code processing method, based on interface files, is that it allows dealing with *incomplete programs*. That is, a module can be processed even if the related modules are still incomplete or completely unavailable. Making dummy definitions of the related modules, defining only their exports, the compiler will produce `.itf` files of them which can be used to process the module in consideration. This can be used also for independent development of different parts of the program, which can then perhaps be performed in parallel by different teams. This also makes possible the early detection of compile-time errors in the module under consideration without having to wait for the code of the related modules to be ready.

Another characteristic of the framework is the automatic incremental (modular) processing of programs. This means that when, e.g., compiling a whole application, the compiler is able to

[1] Thus, a programmer can generate and publish (e.g., for others to see) the interface file at any time by simply running the compiler on the corresponding `.pl` file and granting read permission to the `.itf` file generated. The same programmer can prevent another programmer from overwriting the existing interface file from what may be a possibly intermediate state of the source by simply not granting read permission to the `.pl` file or write permission to the `.itf` file. Furthermore, a precise textual description of the module interface in several documentation formats can be also be generated by simply running the auto documenter [Her99] on the source.

recompile only those modules which need to, automatically following module dependencies starting from the main module of the application. To this end, the `.itf` files in Ciao contain also the information needed to traverse module dependences and to decide if a module has to be recompiled (reprocessed), thus avoiding reading the source code of unchanged modules every time an application is processed. This information includes which modules are imported, which predicates are imported or reexported from each, which external code is included, etc. In summary, it contains the information which compose the premises under which the compilation took place.

An advantage of separate/incremental compilation is that it typically results in a faster development cycle, because it is not necessary to recompile the whole program whenever *any* change is performed on one of the modules composing the program, only those modules affected by the changes will be recompiled. On the down side, less than optimal results (in terms of error detection, degree of optimization, etc., depending on the particular code processing performed) may be obtained compared to an equivalent monolithic compilation, unless some help from the programmer is provided. Note that modularity helps error detection in either case (monolithic/incremental compilation).

These are of course all well known advantages of modular program structure and modular processing, exemplified respectively by languages such as Modula and tools such as Unix `make`. The main novelty is in making this processing generic (so that it is implemented in a consistent way across many tools), highly automated and convenient, and adapting it to a Prolog environment, by virtue of a suitable module system design.

2.2. Operation and implementation of the framework

We now briefly outline the implementation of the Ciao generic code processing framework. It is implemented as a library (called `c_itf`) which exports the following high-order predicate:

```
process_files_from(File, Mode, Type, TreatP, StopP, SkipP, RedoP)
```

This predicate starts a code processing loop with file `File` at its root. `Mode` is the type of code processing, which may be incore compilation, compilation to `.po` (WAM object file), a recursive compilation (started by a `load_compilation_module` directive, see [CH99a]), or any other kind of code processing, as defined by the user of the library. `Type` indicates whether the starting file has to be a module or not (some processes can only be performed on modules, not user files). `TreatP`, `StopP`, `SkipP` and `RedoP` are predicate abstractions (see [CH99b]) which are called in several moments in the code processing loop, and customize it to the required task. They are called with an additional argument which is the identifier of the file being treated at that point, under which relevant data is stored. The framework takes care automatically of low-level tasks including reading into a canonical form the code of the module (if required) and dealing with syntax extensions (operators, expansions, etc.), inclusions of code, redefinition of prolog flags, modules, imports, exports, local definitions, reexports, visibility rules, external code, etc., both in the current module and in the related modules. All this data is easily accessible through the predicates exported by the `c_itf` library.

The code processing loop proceeds as follows: if the related `.itf` of `File` exists and is newer than the source, then the `.itf` file is read, else the source is. If `StopP` succeeds for the file, neither the file is processed nor the loop deepens from it, finishing this branch. Else, the compiler collects the interface data of files from which this file performs imports (its "related files"), reading their `.itf` files (or the source files if those do not exist). Now, if `SkipP` succeeds for the current file, it is not processed further, but this time its branch is followed, entering in the processing loop with the related files. Otherwise, when entering to process the file, if we have read the source file (because the `.itf` file did not exist or was older) we generate a new `.itf` file and call `TreatP` to effectively process the file. If we have read the `.itf` file, we check if any dependence of this file has changed (e.g. an included file has changed, a predicate which this file imported from other module is no longer exported, etc.) and, if it is so, we read the source and proceed as before. Otherwise, we call `RedoP` to verify if we have to process the file anyway, although the `.itf` file does not have to be regenerated. This predicate may, for example, consult additional files which are specific to the particular processing being performed, containing data from the module (as the assertion-cache `.asr` files used by the assertion processing library). Inside `RedoP` it is also possible to perform tasks related with this file which only need the `.itf` information, failing at the end. When the process on this file is over, the loop proceeds with the unprocessed, related files in the same way until there is no more work to do.

3. Compilation of Modular Programs: The Ciao Compiler (`ciaoc`)

We now describe the tasks performed by the Ciao standalone compiler (`ciaoc`),[2] which makes use of the code processing loop presented above, customized for a familiar task: traditional WAM-level compilation. When discussing the functionality, we will use the familiar Unix `cc/make` combination as a reference, since there are many similarities.

The objective of compilation is generally twofold. Firstly, the syntactic analysis typically performed by traditional compilers allows detecting a good number of errors in the program without actually having to run it. Examples are simple syntactic errors, singleton variables, discontiguous clauses, undefined predicates, etc. The second aim of compilation is to generate a lower-level representation of the program which can be executed more efficiently. E.g., in the case of `cc` each source file is compiled into a separate object (`.o`) file containing relocatable machine code. In the case of `ciaoc`, each module is compiled into a separate object (`.po`) file, containing (by default) WAM bytecode. This is the code that will be executed by the Ciao bytecode interpreter at run-time.

The following two sections explain the different tasks of compiling a single module and incrementally compiling a whole program.

3.1. Compiling a Single Module (with the Related Interfaces)

As mentioned before, one of the main benefits of modularity is that it fits very well with the idea of *separate compilation*. The minimal units which can be compiled separately correspond to modules. `cc` itself performs typically only separate compilation of a single file: it is run on a `.c` file and produces a `.o` file. In `ciaoc` this kind of compilation can be performed by selecting the `-c` flag. For example, the command '`ciaoc -c module-a.pl`' performs separate compilation of `module-a` producing `module-a.po`.

As mentioned in Section 2., some information on other modules may be required. In the case of `cc`, the needed information is typically added explicitly to the "module" under consideration (as a result, a reduced amount of error checking can be made).[3] In the case of `ciaoc`, the information needed from related modules to process a module and obtain its compiled version is included in the interface files of these modules, which are automatically managed by the compiler.

[2]As mentioned before, the Ciao compiler is really a library module and can be used from the command line using the `ciaoc` application, from the familiar interactive toplevel shell, etc. While in the discussion we will mention only `ciaoc`, the descriptions given apply equally to the use of the compiler from the toplevel shell or as a library from another program.

[3]In fairness, C is not really modular – we are using it as an example only because the related compilation tools are very well known.

3.2. Incrementally Compiling a Whole Program

The main difference with the previous task is that in this case the objective is not to compile a single module, but rather a set of related modules which compose a whole program and to produce an executable. The compilation starts with the *main* module, which spawns the compilation of other modules from which predicates are being imported. Typically all related modules, i.e., the transitive closure and not just the first level, must be processed. Nevertheless, processing is performed one module at a time, i.e., the compiler processes the code of only one module at each step. In the `cc/make` case this corresponds to writing a `Makefile` (possibly aided by running the `makedepend` command) followed by issuing a `make` command. I.e., the dependencies among modules are first extracted and then based on them the `make` utility determines which modules have to be recompiled.

The Ciao compiler automatically performs this process. Global compilation starts from a module, which is typically the main module of an application. If the module has a recent `.itf` file, it is read in order to decide if the module has to be recompiled and also to be able to follow its dependencies without reading the source. Else, if the source code is newer than the interface file, it is read and stored in memory. Then, all interface files of the modules directly used by this module are accessed. This may involve reading and storing in memory the sources of those modules as well, for those which do not have an up-to-date `.itf` file (later, when those modules are to be processed, their sources are retrieved from memory). Now, the `.itf` file of the current module is generated if needed. This cannot be done before because the interface file includes, among other information, which predicates are imported from other modules. If a module imports "everything" from another module (allowed by the module system design) to produce its interface the other module needs to be accessed to know which predicates it exports.

At this point, we are ready to process the current module, in this case by compiling it to WAM bytecode.[4] But this is done only if the `.po` (object) file does not exist or is older than the current module, or if the compiler finds that current module needs recompilation. An already compiled module which has not changed could need recompilation if, for example, it uses a predicate from another module an the latter module stops defining or exporting it.

The process continues following the dependences with the rest of the modules, compiling only the changed files and using the precompiled `.itf` and `.po` files of the older ones. When all the bytecode object files (with a `.po` extension) are up to date, they are collected and linked by the compiler to build the executable. Note that in a recompilation only the source files which have changed from the previous compilation or which are affected by these same changes have to be read and compiled. From the rest only the `.itf` file is accessed.

Interestingly, we have observed that, if the code for all required modules is available, users tend to choose this compilation scheme over that of the previous section, even when small changes are made, since `ciaoc` automatically determines the modules related to the module under consideration and follows the dependencies among modules deciding which modules require recompilation, without requiring any input from the user. Also note that the compilation method of the previous section can be used in conjunction with a traditional `Makefile` (for example when using Prolog files in the context a larger, multi-language program). However, the Makefile needs to be updated by hand when an interface-related change is made. It is also possible to combine the two approaches and use `ciaoc` from within the `Makefile` to maintain the automatically up to date the Prolog files in the project. This can be done by writing a dummy file which uses all the Prolog files involved.

[4]The compiler also supports external modules written in other languages, and in this case it also automatically calls if needed the right compiler to produce an object file, also possibly regenerating the interface and type conversion code (which is produced automatically from type and mode declarations given in the Ciao assertion language [PBH97] for the external procedures).

4. Errors detected by the compiler

Having discussed the issue of separate compilation and incremental recompilation, in this section we address the second objective of compilation: the early detection of programming errors. This includes also giving warnings about situations which, although strictly correct, are likely a programmer mistake. Here we only deal with the errors detected by `ciaoc`, which performs global analysis at the level of predicate imports and exports. Note that using the Ciao Preprocessor, which performs extensive global dataflow analysis, more (and more involved) error situations can be detected. The most interesting errors detected by the `ciaoc` compiler are:

- *Syntax errors:* This one is almost obliged. Ciao gives the context in which the error is located and the point in the file where this context is located. This allows other tools (e.g., the `emacs` interface) to automatically locate the point in the source file.

- *Unknown directives/declarations:* In Ciao, as in ISO-Prolog [CH99a, PRO94, DEDC96], directives are not conventional queries, so only the ones defined by the language (or in Ciao also by special declarations [CH99a]) are allowed.

- *Redefinition of control constructs:* Although in Ciao it is possible to redefine "builtins", control constructs such as "`,`", "`;`", "`->`", etc. cannot be redefined. Note that a redefinition of said constructs is commonly due to a programmer's mistake, putting a "`.`" in place of a "`,`", as in

```
qsort([X|Xs],L,L2) :-
        partition(Xs,X,Left,Right).
        qsort(Left,L,[X|L1]),
        qsort(Right,L1,L2).
```

- *Illegal imports:* Occur when a module attempts to import a predicate from another module which is not exported by the second.

- *Illegal module qualifications:* In Ciao it is not allowed to bypass module constraints, so this error is issued when a qualified call (as `module:predicate(Args)`) is found to a predicate in a module which is not imported by the current module.

- *Undefined predicate calls (warning):* When compiling a module, in contrast to when compiling a user (nonmodule) file, if a call to a predicate not defined in the module nor imported from another module is found, a warning is given. It is a warning and not an error because the predicate could be defined by a dynamically loaded module. It is possible to explicitly declare the predicates imported from dynamically loaded modules in order to avoid these warnings.

- *Undefined predicate exports (warning):* This is signaled when a predicate in the export list of a module is not defined in the module.

- *Predicates with the same name and different arities (warning):* This warning, from our experience, detects a great number of hard-to-find errors, derived from changing the arity of a predicate in several places but forgetting to change one of the defining clauses. The warning is not issued if the different arity versions are exported by the module, since in that case it is clear that the the different arities exist on purpose. As many Prolog programmers do commonly define predicates with the same name and different arities, this warning can be locally or globally disabled with a Prolog flag.

- *Discontiguous clauses (warning):* As discontiguous clauses of the same predicate are usually the product of mistakes (and also forbidden by ISO-Prolog), this warning is issued when such a case is detected. Nevertheless, these warnings can be switched off by a per-predicate declaration (as in ISO-Prolog) or with a global compiler flag.
- *Singleton variables (warning):* This is the classic test in Prolog compilers, which has proven extremely helpful in finding errors derived from the misspelling of variable names.
- *Local definition of an imported predicate (warning):* This is allowed in Ciao [CH99a], but the warning is signaled because it may be due to a programmer's mistake (as the importation may be of all predicates defined in a module). It can be switched off by adding a `:- redefining` declaration, for a given predicate or also for any predicate.
- *Duplicated importation of a predicate (warning):* This is also allowed in Ciao, but as the previous warning it is signaled because it may be due to a programmer's mistake. It can be switched off in the same way as the previous one.
- *Exporting multifile predicates (warning):* Multifile predicates [CH99a, PRO94, DEDC96] do not need to be exported to be accessible by other modules defining them, thus this warning.
- *Incompatible declarations of multifile predicates:* This error is issued when a multifile predicate is defined differently in several modules (for example, in one is defined *dynamic* while in other is not).

In our experience to date with the Ciao system these messages avoid a great number of otherwise time-consuming errors and result in a much faster development cycle. This situation is enhanced further if the preprocessor is also used. Note however, that some of the choices made in the Ciao module system, which is slightly more strict that the usual Prolog module systems, are necessary to be able to detect many of these errors.

5. Types of Executables Created / Linking Regimes

The Ciao compiler can create different types of executables, depending on the linking regime used for the modules involved. Essentially, modules can be linked into the executable in three ways. If linked *statically*, then the bytecode of the module is added to the executable when building it. This has the advantage that this module does not need to be found elsewhere at execution time but results in a growth in the size of the executable. If linked *dynamically* then the bytecode for the module is not included in the executable but is instead searched for in certain directories defined by a set of *library paths* and loaded at the time when the application starts executing. This has the advantage of a smaller executable size but the required modules must be accessible a run-time for the application to be executed correctly. It is used most frequently with the standard libraries, which can often be assumed to be accessible in the execution environment. Finally if the module is linked *lazily*, then the code for the module is not included in the executable and is instead searched for in the *library paths* directories in the same way as with dynamic linking, but only if during execution the application calls a predicate defined in the module, and at the time of that first call. This has the advantage that the application can start more quickly and that modules which are not used in a particular run do not need to be loaded. Which of these regimes is used for a given module is controlled in a flexible way by associating a loading regime to a certain *path alias* [Swe95], so that a module referred to using that path alias is loaded using the corresponding regime. By default, modules in `library` paths are loaded dynamically, whereas application-specific modules are included in the executable.

There is an additional "executable" type which is not created with the Ciao command-line compiler: Ciao "shell scripts." They are similar to, e.g., the UNIX shell scripts (or the way in which `perl` programs are typically run), but are executed by the Ciao script shell `ciao-shell` which, as mentioned before, is another application which uses the compiler library. The difference with a usual command interpreter script is that the source is compiled (by default; this can be overridden) the first time the application is started, or after a change in the source. This can sometimes be advantageous with respect to creating binary executables for small- to medium-sized programs that are modified often and perform relatively simple tasks. The advantage is that no explicit call to the compiler is necessary, and thus changes and updates to the program imply only editing the source file and invoking again the executable. The disadvantage is that startup of the script (the first time after it is modified) is slower than for an application that has been compiled previously. The interested reader is referred to [Her96, CHV96] for more information and some interesting applications of such scripts.

5.1. Issues in Lazy linking

Lazy linking is implemented by creating stump code which defines each predicate exported by the module as a call to load that module followed by a call to the predicate again. For example, predicate `foo/3` in (lazy-load) module `bar` would be implemented by something like:

```
foo(A,B,C) :- load_lib_lazy(bar), foo(A,B,C).
```

At the time of the last call after the loading, the new definition of predicate `foo/3` has replaced the stump definition, so that the recursive call executes the predicate as if it were loaded from the beginning. Each module to be lazily loaded is replaced in the executable by the code containing the stump predicates of that module, so that the first time an exported predicate of the module is called, the module is loaded. Here the fact that the Ciao module system is strict [CH99a] is instrumental, since if this were not the case all predicates would need a stump definition, not only the exported ones, because external calls to any predicate of the module could happen.

Note that due to the way lazy linking is currently implemented, it is complicated to implement this type of loading for certain types of modules (which are then loaded eagerly in the current implementation). Even in the strict module system of Ciao, in which only exported predicates can be accessed, there are situations in which a module can use the code of the other module without executing an exported predicate of it. One of these situations is when a module exports a dynamic predicate, which can then be accessed by other modules, for example asserting clauses of it. The other situation is when several modules contain the same multifile predicate [CH99a]: when one of these modules calls the multifile predicate all clauses of the predicate need to be present, so the other modules need to be loaded also. These restrictions lead to a transitive relation of "requirement", which defines which other modules need to be loaded when a given module is loaded, in order to safely execute the code. The relation can be computed, as sketched before, as the transitive closure of relation $\mathcal{R}$, defined as the pseudo-clauses:

$\mathcal{R}$(A,B) :- {module A imports a dynamic predicate from module B}.
$\mathcal{R}$(A,B) :- {module A and module B share a multifile predicate}.

Notice that since the main module (the module which contains the starting predicate) is normally linked statically, as in any case it has to be loaded for the executable to start, the above requirement relation dictates that possibly other modules have to be linked the same way. Also, the requirement restriction has to be observed also in the stump clauses: a stump clause of a predicate of a module has to load along with that module other modules required by it.

There are situations in which, in spite of the $\mathcal{R}$ requirement relation keep, there is no possibility that the first module uses code of the second before the second is loaded. This is the case when the manipulation of dynamic predicates of the second module or the calls to shared multifile predicates which occur in the first module are always preceded by executing an exported predicate of the second module, since that way by the time the conflictive call is made the second module will be already loaded. Thus, the compiler provides a mechanism to specify that a module is safe to be lazily loaded, even if a conflict situation is detected by the compiler, because the programmer may know that that is the case. Note that in fact many of these situations can be detected by a more extensive global analysis than that which the current version of the compiler can perform.

6. A Preliminary Experimental Evaluation

The compiler has been fully functional for quite some time now and has been used in a good number of academic and commercial applications, in addition of course to compiling all the components of the Ciao program development environment. While an exhaustive evaluation of the compiler is left as future work, in this section we present some preliminary experimental data on the compiler. All experiments were done in a Pentium-II dual processor at 400 MHz running Linux, shared with a number of other users.

First, and in order to test the incrementality of the compiler, we compare the times required to (re)compile a medium-sized application (in fact, the standalone compiler `ciaoc` itself) depending on the number of files changed. The size of the source comprising this application, including standard libraries, is of about 15,000 lines (53,000 words using UNIX `wc`).

	D	S
From scratch	15.59	18.29
Compiler files	9.77	12.57
Main file	2.82	5.61
No changes	2.62	5.30
No executable generation	2.13	2.13
Executable size (Kb)	167	1105

Table 1: Times (Secs.) to compile `ciaoc`, for different changes in source.

Table 1 shows times in seconds when a dynamic (D) or a static (S) executable is produced. The first row shows the compilation times starting from scratch, that is, no `.itf` (interface) nor `.po` (object) files exist of any source module, including system libraries, i.e., the complete 15,000 lines are compiled. The second row shows the compilation times when system libraries are precompiled, but all compiler-specific source is not. This includes 6 modules, which comprise 52% of the total number of lines and 49% of the total number of words in the whole source. The third row shows the compilation times when only the main module of the application has to be recompiled; system libraries are all precompiled. The main module contains 2.8% of lines and 3.7% of words of the total. Note that in this case, as in the previous case, the fact that many modules are precompiled and unchanged is not known a priori by the compiler, which needs to check that this is so. The fourth row shows the times needed by the compiler to generate the executable file after checking that no source has changed from a previous compilation (all sources are precompiled), i.e., this is the time to link and write out the executable. Finally, the fifth row shows the times under the previous scenario but without generating the executable (thus there is no difference between dynamic or static compilation). The last row shows the size in kilobytes of the executables generated. From the numbers we can conclude that the system is indeed incremental, and incremental compilation is indeed advantageous. Also, we observe that the difference between producing a static or a dynamic executable is related mainly to the relative sizes of the executables, as was to be expected.

We now compare the different linking regimes for executables with regard to compilation time, starting time, size of executable, and memory consumed. The executables used in the tests were: `ciaosh`, the Ciao toplevel shell, a quite large application; `pldiff`, an application to compare Prolog files (à la UNIX `diff`), but disregarding formatting of the code or variable renaming; `wumpus`, an application to solve wumpus world problems [RN95], an AI classic; and `suite`, a set of test programs distributed with the Ciao system (includes several classical benchmarks: queens, fib, etc.).

The results are shown in table 2. Data is given for static (S), lazy (L) and dynamic (D) linking type executables, and for each application. Times are expressed in seconds, sizes in kilobytes.

- T_c is the time that it takes to generate the corresponding executable, given that all source modules have precompiled objects. Note that in the case of lazy linking, a part of this time is spent in compiling the stump code of the lazily-loaded modules. Compilation time is always fastest for dynamic executables, slower for static executables.

- Sz is the size of the executable. Here again the lazy executable is somewhat bigger than its dynamic equivalent because of the inclusion of stump code.

- T_0 is the time to start the application, which includes the loading and partial re-linking of starting object code. Here the lazy executable is fastest, because lazily-loaded modules are not loaded and linked yet. This speedup depends on the amount of lazily-loaded modules compared to the total executable code. Note also that the dynamic executable is a bit slower than the corresponding static executable: in the latter all object code is available just there.

- T_1 is the time spent until work which forces the loading of all modules is done. Surprisingly, in a benchmark this time is smallest for the lazy executable, and in another smaller for the lazy executable than for the dynamic executable. One explanation of this phenomenon might be that by spacing the file accesses, the operating system performs faster each access. Another explanation would be related with fewer page faults and/or more cache hits, since the lazily loaded modules start to execute right after they are loaded.

- M_0 is the memory taken up by the application at time T_0. The amount is quantized because Ciao reclaims memory in chunks, so two equal quantities may represent different real used amounts. The amount is smaller for the lazy executable since there are modules which have not been loaded yet, and which are only represented by their corresponding stump predicates (normally much smaller-sized). The dynamic executable uses slightly more memory than the static executable because the process of finding the modules dynamically consumes some memory.

- Finally, M_1 is the memory taken up by the application at time T_1. Here the lazy executable catches up with or passes the static executable in amount of memory. Also, one of the benchmark shows the dynamic executable using up more memory than the lazy executable. This is because the code which performs the dynamic load in the case of lazily-loaded modules (stump code) is replaced by the object code of the module, while in the dynamic executable it remains in memory. This amount, nevertheless, is minimum, but, as explained before, due to the quantized amounts of memory reclaimed by the system it may appear bigger if it causes to cross a memory usage step.

	T_c	Sz	T_0	T_1	M_0	M_1
`ciaoc`						
S	6.38	1362	1.07	1.19	2704	2704
L	3.92	332	0.84	1.20	2440	2704
D	2.95	245	1.14	1.26	2704	2704
`pldiff`						
S	2.59	352	0.28	0.34	1776	2172
L	2.32	185	0.15	0.32	1644	2172
D	1.95	156	0.29	0.35	1780	2172
`wumpus`						
S	2.12	261	0.21	0.46	1644	2040
L	2.04	194	0.16	0.49	1644	2172
D	1.93	179	0.22	0.48	1780	2172
`suite`						
S	2.25	268	0.22	5.12	1644	3584
L	2.09	171	0.14	5.14	1512	3584
D	1.92	152	0.22	5.13	1780	3716

Table 2: Differences among static, lazy and dynamic executables (Secs., KB).

The results confirm our design intuition that producing static executables is mainly suited for applications which have been tested and which are ready to distributed stand-alone to other environments, where (Ciao) Prolog cannot be assumed to be installed a-priori. Producing dynamic executables, on the other hand, is the best choice for environments where (Ciao) Prolog can be assumed to be installed, and also at development time, when one wants to compile the executable as fast as possible (however, the star during the development cycle is using the toplevel shell from the emacs interface). Finally, lazy executables are very suitable for applications which have parts which may not be used in every run. An example is the Ciao preprocessor, which can perform a very wide variety of tasks, but which in a given run can sometimes be used repeatedly for a very specific task comprising a very small fraction of its overall functionality. The lazy-load executable will automatically load only the small part actually being used, reducing load time and the size of the image in memory.

7. Comparison with Other Systems

We now make some comparisons between the Ciao compiler and the compilers of some other Prolog systems which are popular and of comparable run-time performance. In particular, we discuss differences with SICStus Prolog 3.7.1 (the current version) and Calypso/GNU Prolog 1.0.0 (which has a native-code compiler). Some observations regarding functionality:

- The current SICStus compiler is a very straightforward file-by-file incore compiler which performs virtually no local or inter-module checking at compile-time. It does not keep track of the fact that the compilation of a given file may affect the compilation of others. Also, all operators, expansions, etc. are global, so that they only need to be loaded, but not unloaded. Finally, the module system is supported dynamically, so that there is little overhead at compile-time related to module management.

- The Calypso/GNU Prolog compiler falls in between the SICStus and Ciao compilers. On one hand it does not support modules as SICStus or Ciao and it is not incremental. On the other hand it does perform inter-file analysis to eliminate dead code and detect a number of errors (although, because of the lack of a module system, this can only be done at link time when the source of all the files of an application is present).
- In contrast, the Ciao compiler performs extensive inter-module dependency analysis and error checking, even if only (re)compiling a single module. To this end, the compiler must keep track of all the interfaces and, to support incrementality, read and write interface information and bytecode from and to interface and object files. Also, since in the Ciao module system operators and expansions are local to each module and user file (which we believe is vital to realistically address separate compilation or modular global analysis), they must be loaded and unloaded into the compiler for each file. Also, the module system is implemented mostly statically, which requires program transformations at compile-time (in return for slightly faster execution).
- The executables of SICStus and Ciao are multi-platform, and can be executed in any machine where an "engine" is available. In contrast the Calypso/GNU executables are platform-dependent binary executables. This gives them an advantage in startup time at the expense of portability.
- SICStus and Calypso/GNU only generate static executables, whereas Ciao supports dynamic and lazy-load executables.

We leave performing a comparison of executable size and compilation speed as future work. However, preliminary results show that the size of the *static* executables generated is comparable for Calypso/GNU and Ciao. SICStus does not really build executables, but rather saved states. These saved states were much larger than the executables of Calypso/GNU and Ciao in SICStus version 3.6, but version 3.7 has an excellent *save program* facility which produces very small saved states. The size of these 3.7 saved states when added to the engine size is comparable to those of Calypso/GNU and Ciao.

As for compilation speed, the emphasis to date in the Ciao compiler has not been compilation performance, but rather getting the module system and the resulting incremental and separate compilation capabilities right. The current situation seems to be that the Ciao compiler is of similar performance as the others for large programs and in incremental situations (e.g., recompiling one or a few files in a large executable). However, its seems to be quite a bit slower for small programs or when compiling large programs from scratch, presumably because it must create all the interface and object files. We expect optimizations to improve the raw compiler speed. However, it must also be realized that since the Ciao compiler does much more, it necessarily must take longer to do it. One interesting point is that the compilation time scales well with size. This confirms some of the design decisions aimed at supporting programming-in-the-large.

8. Conclusions

We have presented the Ciao code processing framework, which allows the development of program analysis and transformation tools in a way that is largely orthogonal to the details of module system design. We believe that, for any system which provides a number of code processing tools, having such a framework available and used by all the tools simplifies tremendously the task of making the behavior of the tools consistent. The framework, and the compiler, which is an instance of it, provide separate and global incremental compilation, automatically following module dependencies and recompiling obsolete object and interface files. We have found this to be a very useful feature for any medium- to large-size project. We have presented also some error and warning messages that the compiler can detect statically, which are in practice of great help in avoiding a number of otherwise time-consuming errors. The combination of incremental compilation and additional static error detection results in our experience in a much faster development cycle than with more traditional environments. We have presented the different types of executables that the compiler creates, and through experimental data we have illustrated the different tradeoffs involved. We have found that the tradeoffs are such that there are different uses and environments which make each of these executable types to be the most suitable, and have therefore decided to keep them all as compiler options. Regarding the overall compiler behavior, in our own (admittedly, perhaps biased) experience, we find the Ciao compiler addictive, despite the larger compilation times of this early version, due, among other reasons, to the above mentioned faster development cycle.

9. Acknowledgements

The Ciao abstract machine, the low-level WAM compiler, and some of the library files, are evolutions of the corresponding &-Prolog components, themselves independent evolutions of the excellent corresponding modules in SICStus versions 0.5-0.7. We thank SICS in the name of the logic programming community for allowing public distribution of these components. The Ciao system is a collaborative effort and includes contributions from from members of several institutions, including UPM, SICS, U.Melbourne, Monash U., U.Arizona, Linköping U., NMSU, K.U.Leuven, Bristol U., and Ben-Gurion U. The system documentation and related publications contain more specific credits. The authors would like also to thank the anonymous referees for their constructive comments. The development of Ciao has been funded in part by projects DiSCiPl (ESPRIT LTR 22532 and CICYT TIC 97-1640-CE) and ELLA (CICYT TIC 96-1012-C02-01) (and, previously, by ESPRIT projects "ParForCE" and "ACCLAIM").

References

[BdlBH99] F. Bueno, M. García de la Banda, and M. Hermenegildo. Effectiveness of Abstract Interpretation in Automatic Parallelization: A Case Study in Logic Programming. *ACM Transactions on Programming Languages and Systems*, 21(2):189–238, March 1999.

[CH99a] D. Cabeza and M. Hermenegildo. A New Module System for Prolog. In *ICLP'99 WS on Parallelism and Implementation of (C)LP Systems*. N.M. State U., December 1999.

[CH99b] D. Cabeza and M. Hermenegildo. Higher-order Logic Programming in Ciao. Technical Report CLIP7/99.0, Facultad de Informática, UPM, September 1999.

[CHV96] D. Cabeza, M. Hermenegildo, and S. Varma. The PiLLoW/CIAO Library for INTERNET/WWW Programming using Computational Logic Systems. In *Proceedings of the 1st Workshop on Logic Programming Tools for INTERNET Applications*, JICSLP'96, Bonn, September 1996. Available from `http://clement.info.umoncton.ca/~lpnet`.

[CLI97] The CLIP Group. CIAO System Reference Manual. The Ciao System Documentation Series–TR CLIP3/97.1, Facultad de Informática, UPM, August 1997.

[DEDC96] P. Deransart, A. Ed-Dbali, and L. Cervoni. *Prolog: The Standard*. Springer-Verlag, 1996.

[HBPL99] M. Hermenegildo, F. Bueno, G. Puebla, and P. López. Program Analysis, Debugging and Optimization Using the Ciao System Preprocessor. In *1999 International Conference on Logic Programming*, Cambridge, MA, November 1999. MIT Press.

[Her96] M. Hermenegildo. Writing "Shell Scripts" in SICStus Prolog, April 1996. Posting in `comp.lang.prolog`. Available from `http://www.clip.dia.fi.upm.es/`.

[Her99] M. Hermenegildo. A Documentation Generator for Logic Programming Systems. In *ICLP'99 Workshop on Logic Programming Environments*. N.M. State University, December 1999.

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

[PBH97] G. Puebla, F. Bueno, and M. Hermenegildo. An Assertion Language for Debugging of Constraint Logic Programs. In *Proceedings of the ILPS'97 Workshop on Tools and Environments for (Constraint) Logic Programming*, October 1997. Available from `ftp://clip.dia.fi.upm.es/pub/papers/assert_lang_tr_discipldeliv.ps.gz` as technical report CLIP2/97.1.

[PRO94] International Organization for Standardization, National Physical Laboratory, Teddington, Middlesex, England. *PROLOG. ISO/IEC DIS 13211 — Part 1: General Core*, 1994.

[PRO95] International Organization for Standardization, National Physical Laboratory, Teddington, Middlesex, England. *PROLOG. Working Draft 7.0 X3J17/95/1 — Part 2: Modules*, 1995.

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

[RN95] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, Englewood Cliffs, New Jersey, 1995.

[Swe95] Swedish Institute of Computer Science, P.O. Box 1263, S-16313 Spanga, Sweden. *Sicstus Prolog V3.0 User's Manual*, 1995.