An Intro to Execution Models and Environments Through the Ciao System

Manuel Hermenegildo
herme@fi.upm.es
http://www.clip.dia.fi.upm.es/herme

IMDEA Software
Tech. University of Madrid
U. of New Mexico

The UPM work presented is a joint effort with members of the CLIP group at the UPM School of Computer Science and IMDEA Software including: Francisco Bueno, Daniel Cabeza, Manuel Carro, Amadeo Casas, Pablo Chico, Jess Correas, María José García de la Banda, Manuel Hermenegildo, Pedro López, Mario Mndez, Edison Mera, Jos Morales Jorge Navas, and Germán Puebla.
Motivation and Approach

Objective (one of the main (and hardest) challenges in CS):

- Facilitating the process of building complex software components/systems, in as short a time as possible,
- while obtaining guaranteed safety, reliability, and efficiency.

Approach:

- Develop (CLP-based) next-generation, high-level, multiparadigm programming languages and execution models.
- Develop improved progr. devel. environments, which perform as part of compilation:
  - Verification / debugging, which can detect bugs or offer guarantees of safety, reliability, and efficiency.
  - Optimization (optimized compilation, parallelization, ...).

Use throughout techniques that are at the same time rigorous and practical.

Apply in a concrete system, with real users –reality check! (In our case: Ciao/CiaoPP system.)
Modular, Evolvable Design

- Ciao is built in layers over a small, pure (LP-based) *kernel*.
  - Allows designing *syntactic and semantic extensions* in a simple, flexible, and scalable way.
  - Also, building small, fast executables and embeddability (non-needed parts of the language and libraries are not included).

- Fundamental enabler –its module/class system:
  - Designed from the ground up to be *extensible* and *analysis-friendly*.
  - Most language features (loops, conditionals, functions, ...) are not built-in, but rather in libraries.
  - Language extension libraries (“packages”) affect only the modules in which they are loaded (e.g., operators and expansions are local to modules, etc.).

  → Allows modular program development, separate/incremental compilation.
  → Allows extensive global analysis for detecting errors and optimizing code.
  → Support for programming “in the large.”
Multiparadigm

- **Logic programming:**
  - Certainly ISO-Prolog (one of the best Prologs!) — but *via a library*; and also:
    - Pure LP.
    - Various comp. rules: breadth-first, iterative-deepening, Andorra, *tabling*, etc.

- **Functional programming:**
  - Function definitions and function calls and functional syntax for predicates.
  - *Higher-order* and *lazyness* for functions and predicates.

- **Constraint programming:** clpr, clpq, fd, Leuven CHR.

- **Objects:** a naturally embedded notion of classes and objects.

- **Concurrency, parallelism, distributed execution.**

- **Imperative features:** mutable data struct., assignment, loops, cases, etc.

- **Assertion language**, consistent across paradigms; with many uses!

+ many other packages: types, records, DCGs, negations, appl.-specific languages,
  ...
The Ciao Module System

Ciao implements a module system [10] which meets a number of objectives:

- High extensibility in syntax and functionality: allows having pure logic programming and many extensions.
- Makes it possible to perform modular (separate) processing of program components (without “makefiles”).
- Greatly enhanced error detection (e.g., undefined predicates).
- Facilitates (modular) global analysis.
- Support for meta-programming and higher-order.
- Predicate based-like, but with functor/type hiding.

while at the same time providing:

- High compatibility with traditional standards (Quintus, SICStus, ...).
- Backward compatible with files which are not modules.
Defining modules and exports

:- module(module_name, list_of_exports, list_of_packages).

Declares a module of name `module_name`, which exports `list_of_exports` and loads `list_of_packages` (packages are syntactic and semantic extensions).

Example: `:- module(lists, [list/1, member/2], [functions]).`

Examples of some standard uses and packages:

- `:- module(module_name, [exports], []).`  
  ⇒ Module uses (pure) kernel language.

- `:- module(module_name, [exports], [packages]).`  
  ⇒ Module uses kernel language + some packages.

- `:- module(module_name, [exports], [functions]).`  
  ⇒ Functional programming.

- `:- module(module_name, [exports], [assertions, functions]).`  
  ⇒ Assertions (types, modes, etc.) and functional programming.
Defining modules and exports (Contd.)

(ISO-)Prolog:

\[- \text{module}(\text{module\_name}, [\text{exports}], [\text{iso}]).\]

⇒ Iso Prolog module.

\[- \text{module}(\text{module\_name}, [\text{exports}], [\text{classic}]).\]

⇒ “Classic” Prolog module

(ISO + all other predicates that traditional Prologs offer as “built-Ins”).

Special form:

\[- \text{module}(\text{module\_name}, [\text{exports}]).\]

Equivalent to:

\[- \text{module}(\text{module\_name}, [\text{exports}], [\text{classic}]).\]

⇒ Provides compatibility with traditional Prolog systems.
Defining modules and exports (Contd.)

- Useful shortcuts:
  
  ```prolog
  :- module(_, list_of_exports).
  ```

  If given as "_" module name taken from file name (default).

  Example: :- module(_, [list/1, member/2]). (file is lists.pl)

  ```prolog
  :- module(_, _).
  ```

  If "_" all predicates exported (useful when prototyping / experimenting).

- “User” files:

  Traditional name for files including predicates but no module declaration.
  Provided for backwards compatibility with non-modular Prolog systems.
  Not recommended: they are problematic (and, essentially, deprecated).
  Much better alternative: use :- module(_, _). at top of file.

  As easy to use for quick prototyping as “user” files.

  Lots of advantages: much better error detection, compilation, optimization, ...

Importing from another module

Using other modules in a module:

- use_module(filename).
  Imports all predicates that filename exports.
- use_module(filename, list_of_imports).
  Imports predicates in list_of_imports from filename.
- ensure_loaded(filename). —for loading user files (deprecated).

When importing predicates with the same name from different modules, module name is used to disambiguate:

- module(main,[main/0]).
- use_module(lists, [member/2]).
- use_module(trees, [member/2]).

main :-
    produce_list(L),
    lists:member(X,L),
    ...

Some more specific characteristics [10]:

- Syntax, flags, expansions, etc. are local to modules.
- Compile-time and run-time code is clearly separated (e.g., expansion code is compile-time and does not go into executables).
- “Built-ins” are in libraries and can be loaded into and/or unloaded from the context of a given module.
- Dynamic parts are more isolated.
- Directives are not queries.
- Richer treatment of meta-predicates and higher-order.
- The entry points to modules are statically defined.
- Module qualification used only for disambiguating predicate names.
- All module text must be available or in related parts.

A resulting notion: packages (see later).
Compiler

- Compiler (standalone –ciaoc– or embedded in top level):
  - Follows dependencies and (re-)compiles modules separately automatically (no need for make- or project-style files).
  - Creates small, standalone executables and libraries.
  - Can also be used in conjunction with make-style tools.

- Produces several types of executables:
  - *Static*: all libraries included.
  - *Dynamic*: libraries loaded dynamically at run-time.
  - *Lazy load*: as dynamic, but libraries are loaded only if actually used.

The modes above can be mixed.

- Also, Ciao “scripts”:
  - Useful for small applications, CGI scripts, etc.
  - Used to make installation procedures platform-independent.
  → Support for programming “in the small.”
Advanced Development Environment

- *Emacs* and *eclipse* versions.
- Top-level, source debugger, standalone optimizing compiler, script interpreter, ...
- Autodocumenter, large set of libraries, ...
- Preprocessor (*ciaopp*):
  - Input: program, optionally w/assertions declaring properties such as types, modes, det, nf, sizes, cost, resources, etc.
  - Output: *error/warning messages* + *transformed program*, with
    - Results of static checking of assertions / error detection / verification. (and certificates for Abstraction Carrying Code).
    - Assertion run-time checking code.
    - High-level optimizations (specialization, slicing, parallelization).
    - Results of analysis (as assertions): used for low-level optimizations.
  - Technology: modular polyvariant abstract interpretation/specialization.
Some Features of The Ciao Development Environment

- Provides:
  - Incremental syntax highlighting of source code.
  - Direct access to on-line documentation (help and completion on what the cursor is on).
  - Direct, interactive access to compiler, top-level, preprocessor, etc.
  - Location of errors from compiler (and preprocessor) on source code.
  - Source code debugging.
  - Direct access to *auto-generation* of documentation.
  - Menu-driven access + also keyboard shortcuts and toolbar.
  - User extensible.
  - Plus many other features!

- Built as a powerful extension of *emacs*.

- Also eclipse plugins have been developed (as contribs).
Autodocumenter: LPdoc

Uses:

- All the information that the compiler has.
- Assertions.
- Comment declarations:
  
  ```prolog
  :- comment(title,"Complex numbers library").
  :- comment(summary,"Provides an ADT for complex numbers.").
  ```

- Markup language, close to \LaTeX/texinfo.
  With indices, references, figures, ...
Debugging in Ciao

- The traditional interface to the “Byrd–box” debugger is available:

  ![Diagram of debugger interface]

  ```prolog
  + 13 7  Call: T user:descendant(dani,_123) ?
  ```

- In addition, source-level tracking of the debugging process is supported:
  - Simultaneous visualization of tracing messages and byrd-box ports in the source code.
  - Placing break-points directly on the source code.

- Debugging modes can be toggled on a per-module basis:
  - `debug_module/1`, `nodebug_module/1`, `debug_module_source/1`

- Easiest: use the Emacs environment.

- The debugger is also a library → Debugging also available in standalone executables! (see later).
Multiparadigm

- **Logic programming:**
  - Certainly ISO-Prolog (one of the best Prologs!) –but *via a library*; and also:
    - Pure LP.
    - Various comp. rules: breadth-first, iterative-deepening, Andorra, *tabling*, etc.

- **Functional programming:**
  - Function definitions and function calls and functional syntax for predicates.
  - *Higher-order* and *lazyness* for functions and predicates.

- **Constraint programming:** clpr, clpq, fd, Leuven CHR.

- **Objects:** a naturally embedded notion of classes and objects.

- **Concurrency, parallelism, distributed execution.**

- **Imperative features:** mutable data struct., assignment, loops, cases, etc.

- **Assertion language**, consistent across paradigms; with many uses!

+ many other packages: types, records, DCGs, negations, appl.-specific languages, ...

M. Hermenegildo – Exec. models and Envs. through Ciao
Using Other Computation Rules

- Libraries which replace the default depth-first, left to right computation rule of Ciao (and Prolog).
- Compile-time transformations (“Compiling Control” techniques).
- Useful in search problems when a complete proof procedure is needed. (e.g., for teaching pure logic programming!)
- Computation rules currently implemented:
  - Breadth-first (bf and ’bf/bfall’ packages).
  - Iterative-deepening (id package).
  - Depth-First search with limited depth (id package).
  - Fuzzy LP. Mycin.
  - “And-fair” breadth-first (bf/af package).
  - Andorra (in Beta).
  - Tabling (in Beta).
- pure package + bf (or id etc.) ideal for teaching LP!
CiaoPP: The Ciao System Preprocessor

- CiaoPP is a preprocessor for the standard Ciao clause-level compiler.
- Performs error detection, verification, and source-to-source transformations:
  - Input: logic program (optionally w/assertions & syntactic extensions).
  - Output: *error/warning messages* + *transformed logic program*, with
  - Results of static checking of assertions / verification.
    (and certificates for Abstraction Carrying Code).
  - Assertion run-time checking code.
  - High-level optimizations (specialization, slicing, parallelization).
  - Results of analysis (as assertions): used for low-level optimizations.
- Underlying technology:
  - Modular polyvariant abstract interpretation.
  - Modular abstract multiple specialization.
- See specific tutorial on the CiaoPP system.
Some Other Features

- “Industry standard” performance, with upcoming highly optimizing compiler.
- (Semi-automatic) interfaces to and from Ciao for C, Java, tcl-tk, etc.
- Ciao engine can be included as a (possibly dynamic) library in applications.
- Support for concurrency, parallelism, and distributed execution.
- Persistent predicate-based interface to relational databases.
- Infinite precision integers (“bignums”) and floating point numbers.
- Many others...
Why Ciao?

- Why is the system called “Ciao”?
- It is one hand an acronym:
  - CIAO: Constraint Programming with Independent And + Or parallelism.
- But the name also represents the *spirit* of the system:
  - Ciao is an interesting word that means *both Hello* and *Goodbye*.
  - “Ciao Prolog:”
    - is aimed at introducing programmers to Prolog and LP/CLP
      – the “Hello Prolog” part,
    - but it also represents really a new-generation programming language and environment (with FP, HO, assertions, global analysis, objects, ...)
      – the “Goodbye Prolog” part.
Some Members of The Ciao Forge

Ciao is really a widely distributed collaborative effort:

- Directly within the CLIP Group:

- Plus lots of contributors worldwide:
Downloading the systems

- Downloading Ciao, CiaoPP, LPdoc, and other CLIP software:
  - Standard distributions:
    - http://www.clip.dia.fi.upm.es/Software
  - Some betas (in testing or completing docs – ask webmaster for info) in:
    - http://www.clip.dia.fi.upm.es/Software/Beta
  - Mail list stored in
    - http://www.clip.dia.fi.upm.es/Mail/ciao-users/
  - Please contact us for SVN access.

- User’s mailing list:
  - ciao-users@clip.dia.fi.upm.es

Subscribe by sending a message with only subscribe in the body to
  - ciao-users-request@clip.dia.fi.upm.es
Some Bibliography on Ciao, CiaoPP, and LPdoc


