Computational Logic: (Constraint) Logic Programming *Theory, practice, and implementation*

The *Ciao* Programming Environment and Multiparadigm Programming

The following people have contributed to this course material:

Manuel Hermenegildo (editor), Francisco Bueno, Manuel Carro, Germán Puebla, Pedro López, and Daniel Cabeza, Technical University of Madrid, Spain

(These slides correspond to a tutorial given at CL2000)

Introduction: The Ciao Program Development System

- Ciao is a next-generation *multiparadigm* programming environment – features:
  - Free software (GNU LGPL license).

Introduction: The Ciao Program Development System

- Ciao is a next-generation *multiparadigm* programming environment – features:
  - Free software (GNU LGPL license).
  - Designed to be *extensible* and *analysis-friendly*.

Introduction: The Ciao Program Development System

- Ciao is a next-generation *multiparadigm* programming environment – features:
  - Free software (GNU LGPL license).
  - Designed to be *extensible* and *analysis-friendly*.
  - Support for several paradigms:
    - pure (*no "built-ins"*), but very powerful and extensible *kernel*.
    - pure LP, ISO-Prolog, functions, higher-order, constraints, objects, ...
    - concurrency, parallelism, distributed execution, ...

Introduction: The Ciao Program Development System

- Ciao is a next-generation *multiparadigm* programming environment – features:
  - Free software (GNU LGPL license).
  - Designed to be *extensible* and *analysis-friendly*.
  - Support for several paradigms:
    - pure (*no "built-ins"*), but very powerful and extensible *kernel*.
    - pure LP, ISO-Prolog, functions, higher-order, constraints, objects, ...
    - concurrency, parallelism, distributed execution, ...
  - Support for *programming in the large*:
    - robust module/object system, separate/incremental compilation, ...
    - "industry standard" performance.
    - (semi-automatic) interfaces to other languages, databases, etc.
    - assertion language, automatic static inference and checking, autodoc, ...

Introduction: The Ciao Program Development System

- Ciao is a next-generation *multiparadigm* programming environment – features:
  - Free software (GNU LGPL license).
  - Designed to be *extensible* and *analysis-friendly*.
  - Support for several paradigms:
    - pure (*no "built-ins"*), but very powerful and extensible *kernel*.
    - pure LP, ISO-Prolog, functions, higher-order, constraints, objects, ...
    - concurrency, parallelism, distributed execution, ...
  - Support for *programming in the large*:
    - robust module/object system, separate/incremental compilation, ...
    - "industry standard" performance.
    - (semi-automatic) interfaces to other languages, databases, etc.
    - assertion language, automatic static inference and checking, autodoc, ...
  - Support for programming *in the small*:
    - scripts, small (static/dynamic/lazy-load) executables, portability, ...

Introduction: The Ciao Program Development System

- Ciao is a next-generation *multiparadigm* programming environment – features:
  - Free software (GNU LGPL license).
  - Designed to be *extensible* and *analysis-friendly*.
  - Support for several paradigms:
    - pure (*no "built-ins"*), but very powerful and extensible *kernel*.
    - pure LP, ISO-Prolog, functions, higher-order, constraints, objects, ...
    - concurrency, parallelism, distributed execution, ...
  - Support for *programming in the large*:
    - robust module/object system, separate/incremental compilation, ...
    - "industry standard" performance.
    - (semi-automatic) interfaces to other languages, databases, etc.
    - assertion language, automatic static inference and checking, autodoc, ...
  - Support for programming *in the small*:
    - scripts, small (static/dynamic/lazy-load) executables, portability, ...
  - Advanced programming environment (with, e.g., automatic access to docs).

A Parenthesis: Why Ciao?

- Why is the system called Ciao?
- It was at some point actually an acronym:

A Parenthesis: Why Ciao?

- Why is the system called Ciao?
- It was at some point actually an acronym:
  - *CIAO*: A **C**onstraint Programming Language with, **I**ndependent **A**nd + **O**r parallelism.

A Parenthesis: Why Ciao?

- Why is the system called Ciao?
- It was at some point actually an acronym:
  - *CIAO*: A **C**onstraint Programming Language with, **I**ndependent **A**nd + **O**r parallelism.
- But the word itself represents the spirit of the system:
  - Ciao is an interesting word that means *both Hello* and *Goodbye*.
  - So, the connotation of “Ciao Prolog” is that:
    - It is aimed at introducing programmers to Prolog –the “Hello Prolog” part,
    - but it also then goes much beyond (with CLP, FP, HO, Objects, ...) –the “Goodbye *traditional* Prolog” part.

Introduction: The Ciao Program Development System (Contd.)

- Components of the environment (independent, written in Ciao):

  `ciaosh`: Standard top-level shell.
  `ciaoc`: Standalone compiler.
  `ciao-shell`: Script interpreter.
  `lpdoc`: Documentation generator (info, ps, pdf, html, ...).
  `lpmake`: Like make but with all Ciao behind.
  `ciaopp`: Preprocessor (assertion checker/optimizer/parallelizer...).

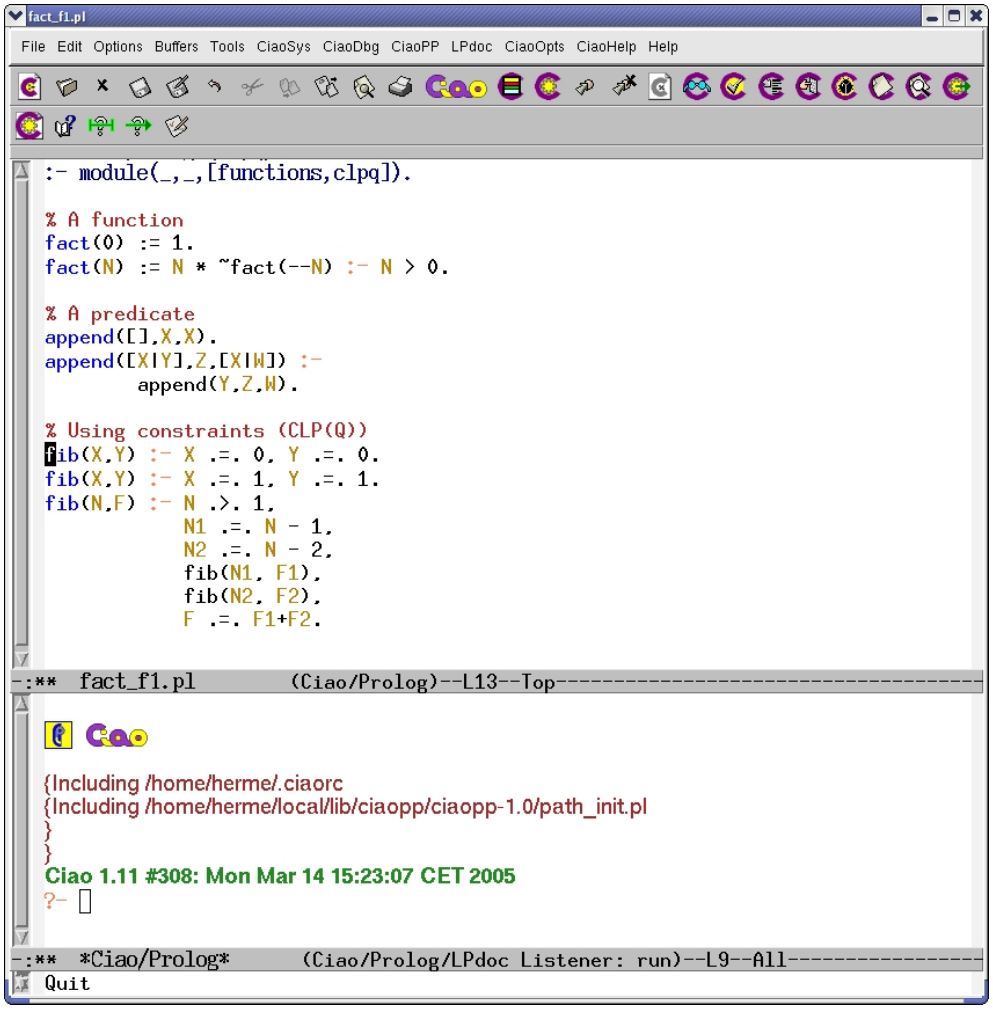
  + Many libraries:
    - ◇ Records (argument names).
    - ◇ Persistent predicates (automatically updated and stored in permanent media).
    - ◇ Transparent interface to databases.
    - ◇ Interfaces to C, Java, tcl-tk, etc.
    - ◇ Distributed execution.
    - ◇ Interface to current Internet standards and protocols (e.g., the PiLLoW library: HTML, forms, http protocol, VRML generation, etc.), ...

The Ciao Development Environment

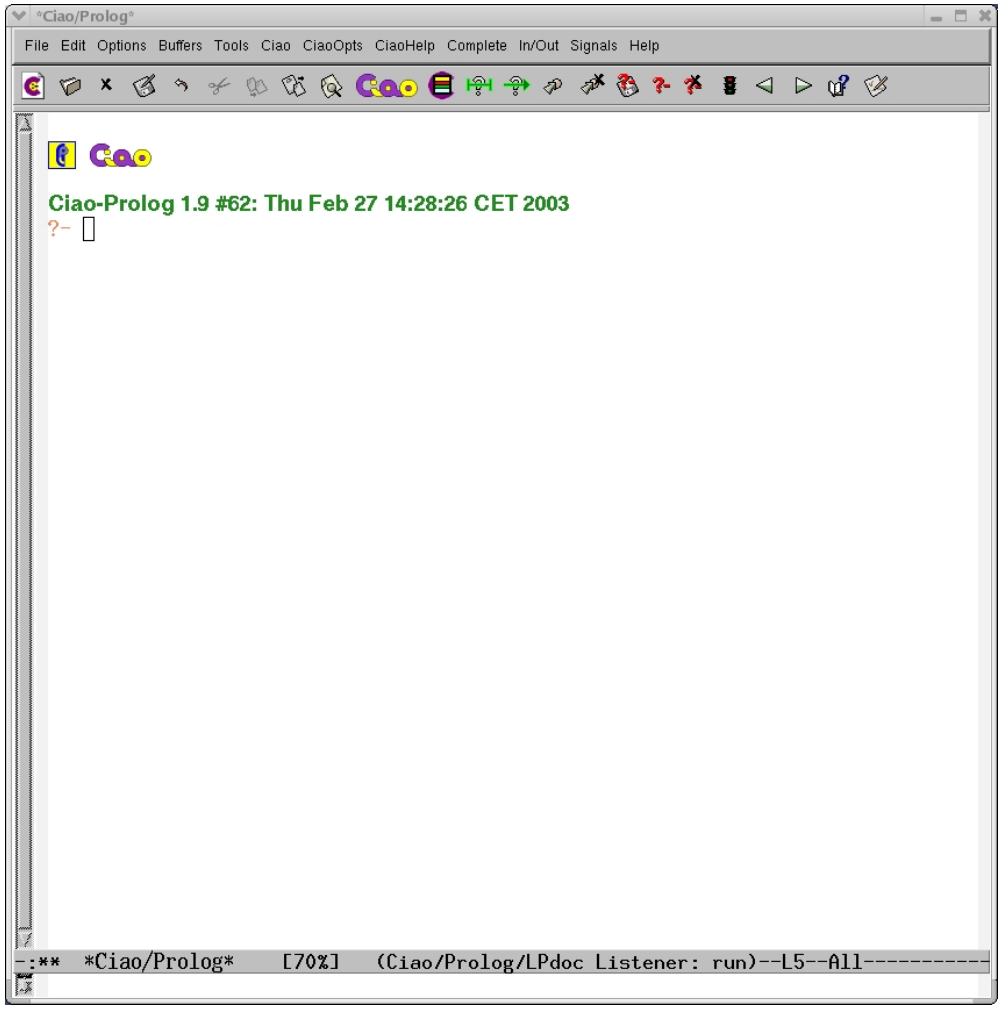
The Ciao Development Environment

- 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 pluings have been developed (as contribs).

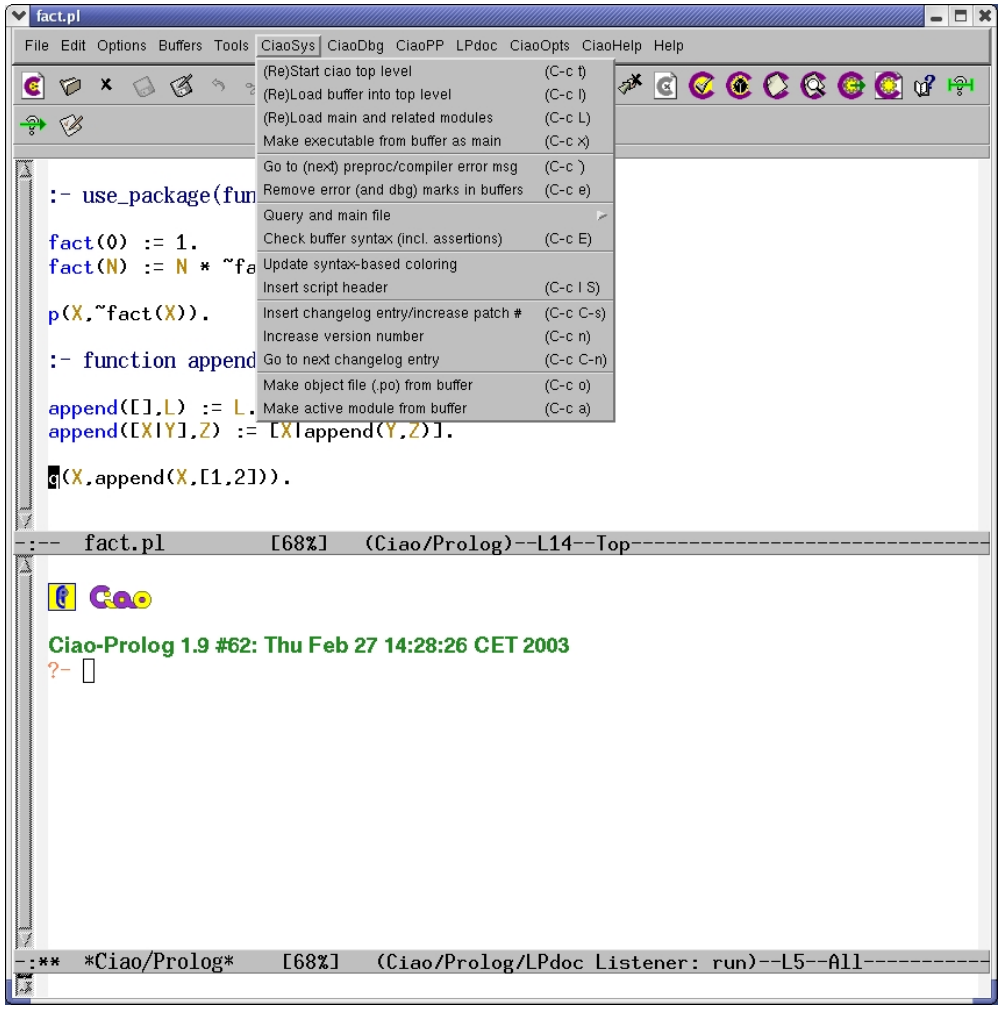
Ciao Prog. Environment: The Graphical Environment



Ciao Prog. Environment: The Top Level



Ciao Prog. Environment: Main Menu (compilation, error location, etc.)



Ciao Program Load

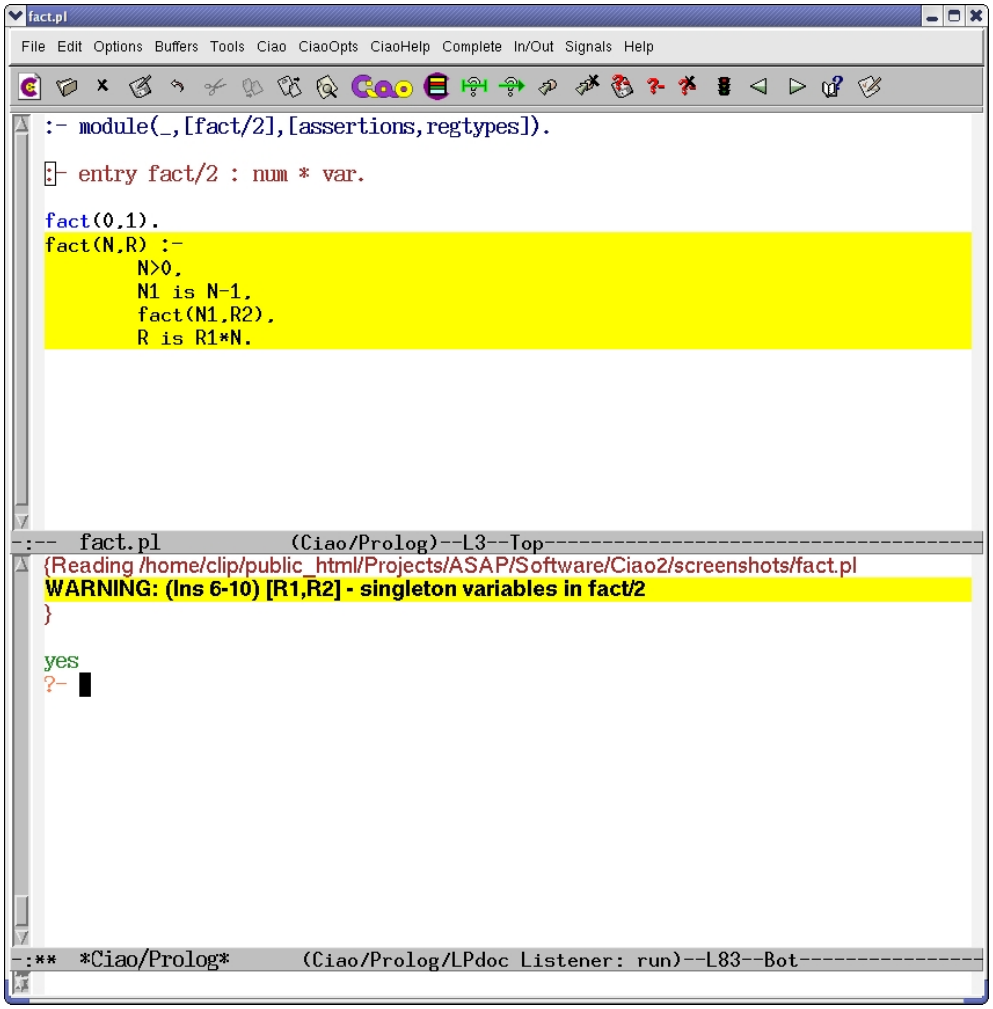
- Most traditional ("Edinburgh") program load commands can be used.
- But more modern primitives available which take into account module system. *Same commands used as in the code inside a module:*
  - `use_module/1` – for loading modules.
  - `ensure_loaded/1` – for loading user files.
  - `use_package/1` – for loading packages (see later).
- In summary, top-level behaves essentially like a module.

- In practice, *done automatically within graphical environment*:
  - Open the source file in the graphical environment.
  - Edit it (with syntax coloring, etc.).
  - Load it by typing `C-c l` or using menus.
  - Interact with it in top level.

Ciao System Menu (Partial)

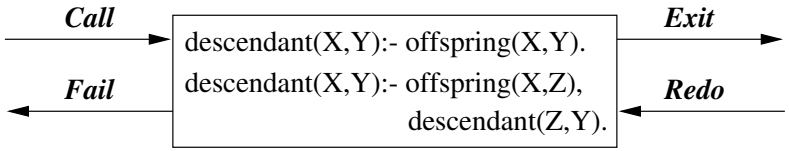
(Re)Start Ciao top level	(C-c t)
(Re)Load buffer into top level	(C-c l)
(Re)Load and check buffer into top level	(C-c f)
(Re)Load main and related modules	(C-c L)
Make executable from buffer as main	(C-c x)
Go to (next) preproc/compiler error msg	(C-c ‘)
Remove error (and dbg) marks in buffers	(C-c e)
Set default query	(C-c q)
Call default query	(C-c Q)
Clear default query	
(Un)Set main module	(C-c s)
Update syntax-based coloring	
Insert script header	(C-c I S)
Make object file (.po) from buffer	(C-c o)
Make active module from buffer	(C-c a)

Ciao Prog. Environment: Error Location



Debugging in Ciao

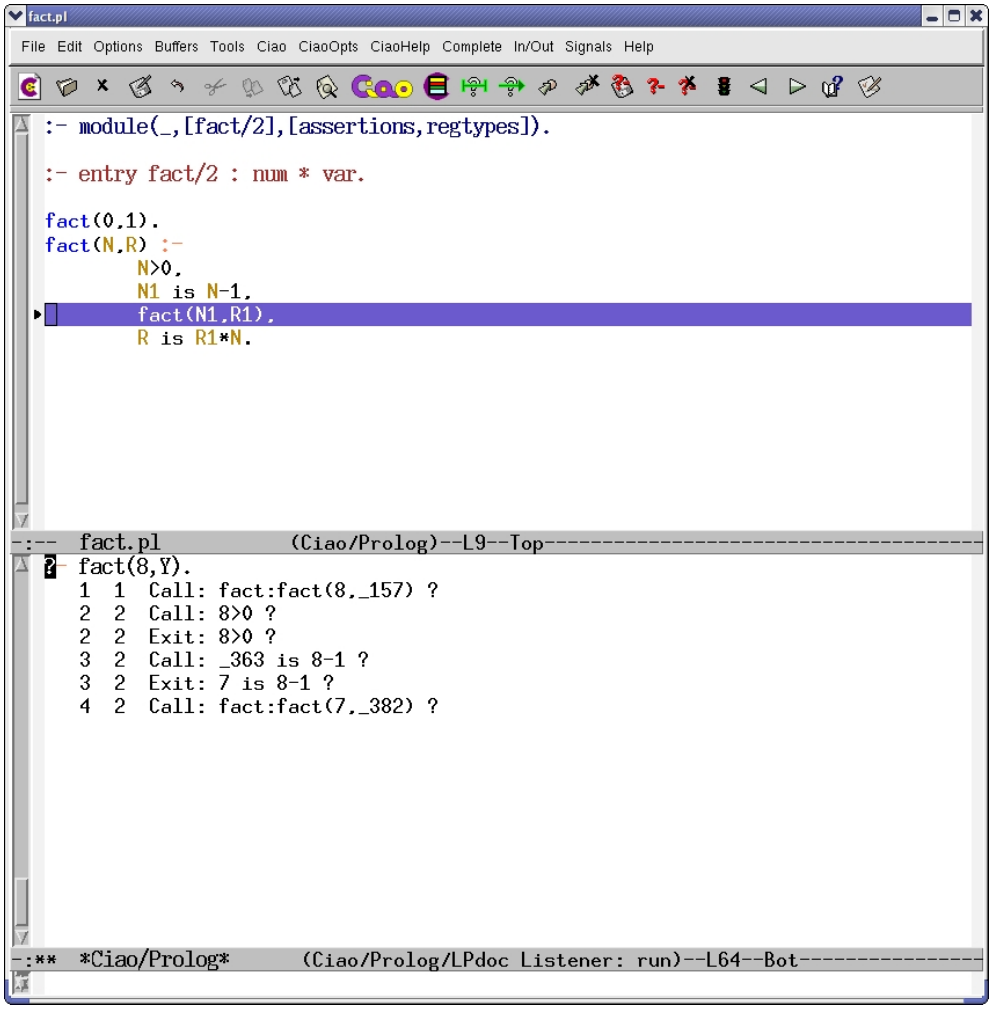
- The traditional interface to the “Byrd–box” debugger is available:




```
+  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 $\rightarrow$
  Debugging also available in standalone executables! (see later).

Ciao Prog. Environment: The Source Debugger



Ciao Debugging Menu (Partial)

(Un)Debug buffer source	(C-c d)
Select debug mode	(C-c m)
Select multiple buffers for debug	(C-c M-m)
Set breakpoint on current literal pred symb	(C-c S b)
Remove breakpoint from current literal	(C-c S v)
Remove all breakpoints	(C-c S n)
Redisplay breakpoints	(C-c S l)
Toggle debug mode (jump to bkp or spypt)	(C-c S d)
Toggle trace mode	(C-c S t)
(Re)Load region (for debug)	(C-c r)
(Re)Load predicate (for debug)	(C-c p)

Breakpoints

- Breakpoints are associated to literals rather than to predicates (as spypoints are).
- Spypoints trace every goal of the predicate, breakpoints only those arising from the selected literal in the program source.
- Information associated with a breakpoint:
  - ◇ File name.
  - ◇ Predicate name.
  - ◇ Start and end lines of the clause.
  - ◇ Number of literal in the clause & actual line of the literal.
- Set/unset/list breakpoints:
  - ◇ `breakpt/6`
  - ◇ `nobreakpt/6`
  - ◇ `nobreakall/0`
  - ◇ `list_breakpt/0`
- Easiest to use from the Emacs environment.

Using the Debugger

- Activate debugging mode (for the module) and load the code (from the file).
- Toggle debugging modes.
- Set/unset/list breakpoints.
- Plus the classical spy-points.

Everything transparent to the user within Emacs!

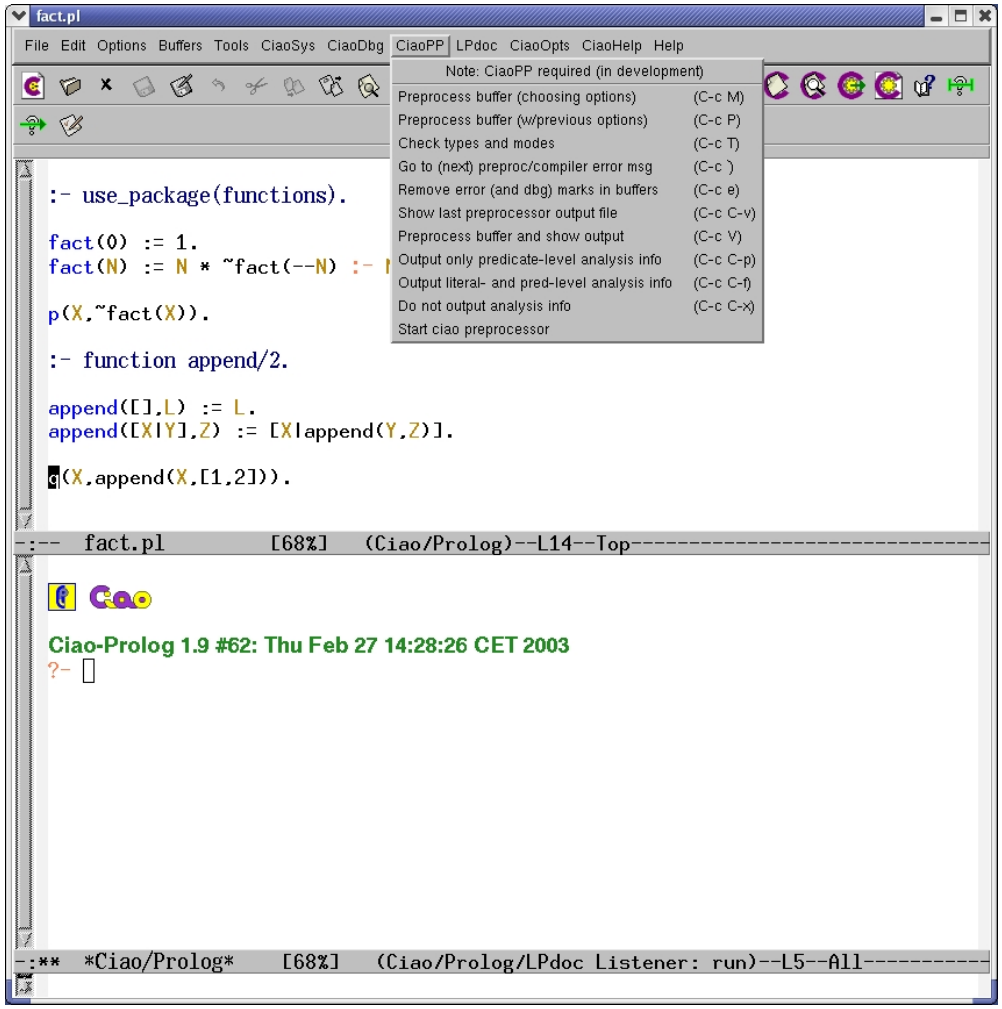
- Plus the usual menu commands of the tracer.
- Additionally:
  - Source-debugging also useful outside Emacs:

```
            In /home/clip/ciao/dbgex.pl (5-9) descendant-1
+  13  7    Call: T user:descendant(dani,_123) ?
```

  - Debugger works also in (stand-alone) compiled executables:

```
:- use_package(debug).
```

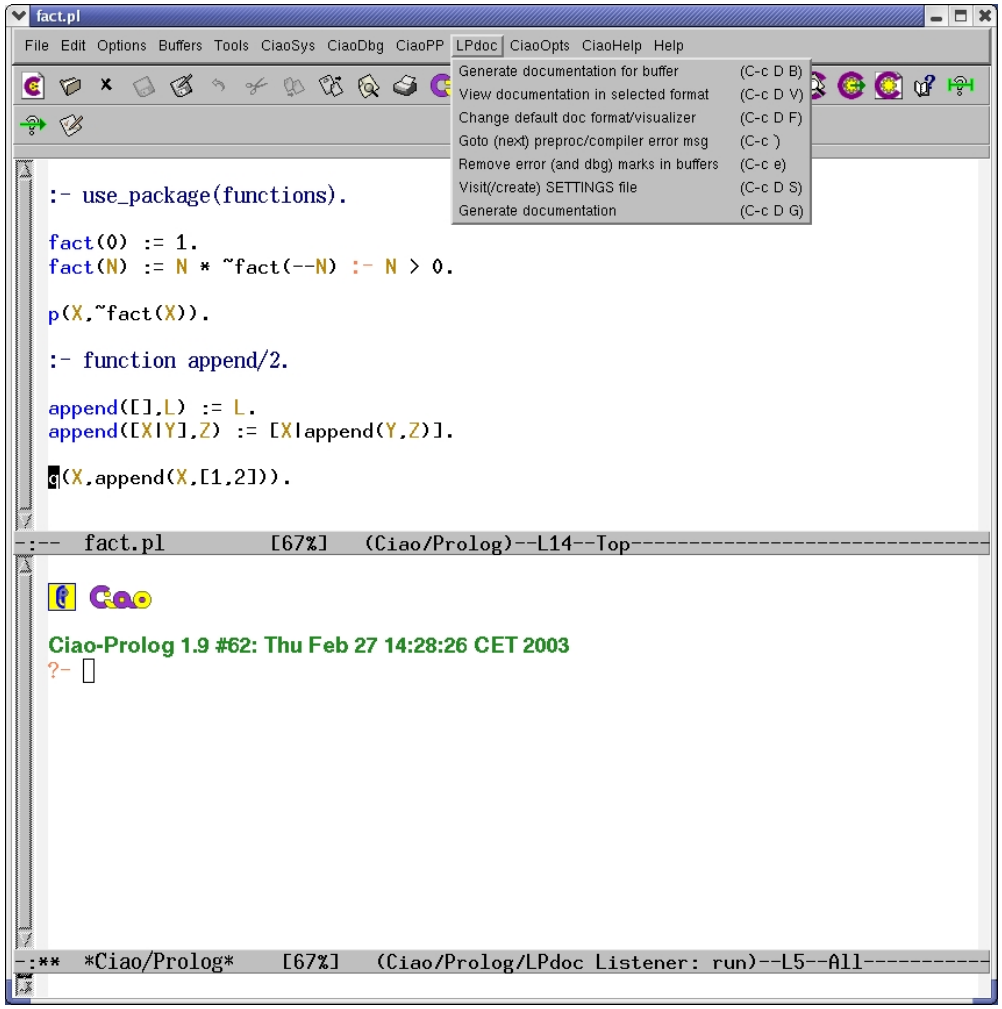
Ciao Prog. Env.: CiaoPP (debugging, optimization, verification, ...)



Ciao Preprocessor Menu (Partial)

Analyze buffer	(C-c A)
Check buffer assertions	(C-c T)
Optimize buffer	(C-c O)
Browse analysis/checking/optimizing options	(C-c M)
Go to (next) preproc/compiler error msg	(C-c ‘)
Remove error (and dbg) marks in buffers	(C-c e)
Show last preprocessor output file	(C-c C-v)
Start Ciao preprocessor	

Ciao Prog. Environment: Menu for Generating Documentation



Ciao LPdoc Menu (Partial)

Generate documentation for buffer	(C-c D B)
View documentation in selected format	(C-c D V)
Change default doc format/visualizer	(C-c D F)
Goto (next) preproc/compiler error msg	(C-c ‘)
Remove error (and dbg) marks in buffers	(C-c e)
Visit(/create) LPSETTINGS.pl file	(C-c D S)
Generate documentation	(C-c D G)

Set version control for file	(C-c C-a)
Insert changelog entry/increase patch #	(C-c C-s)
Increase version number	(C-c n)
Go to next changelog entry	(C-c C-n)

Ciao Customization Menu (Partial)

```
Customize all Ciao environment settings
```

```
Customize all Ciao system environment settings
Set Ciao toplevel executable
Set Ciao toplevel args
```

```
Set Ciao library path
```

```
Customize all CiaoPP environment settings
Set Ciao Preprocessor executable
Set Ciao Preprocessor executable args
```

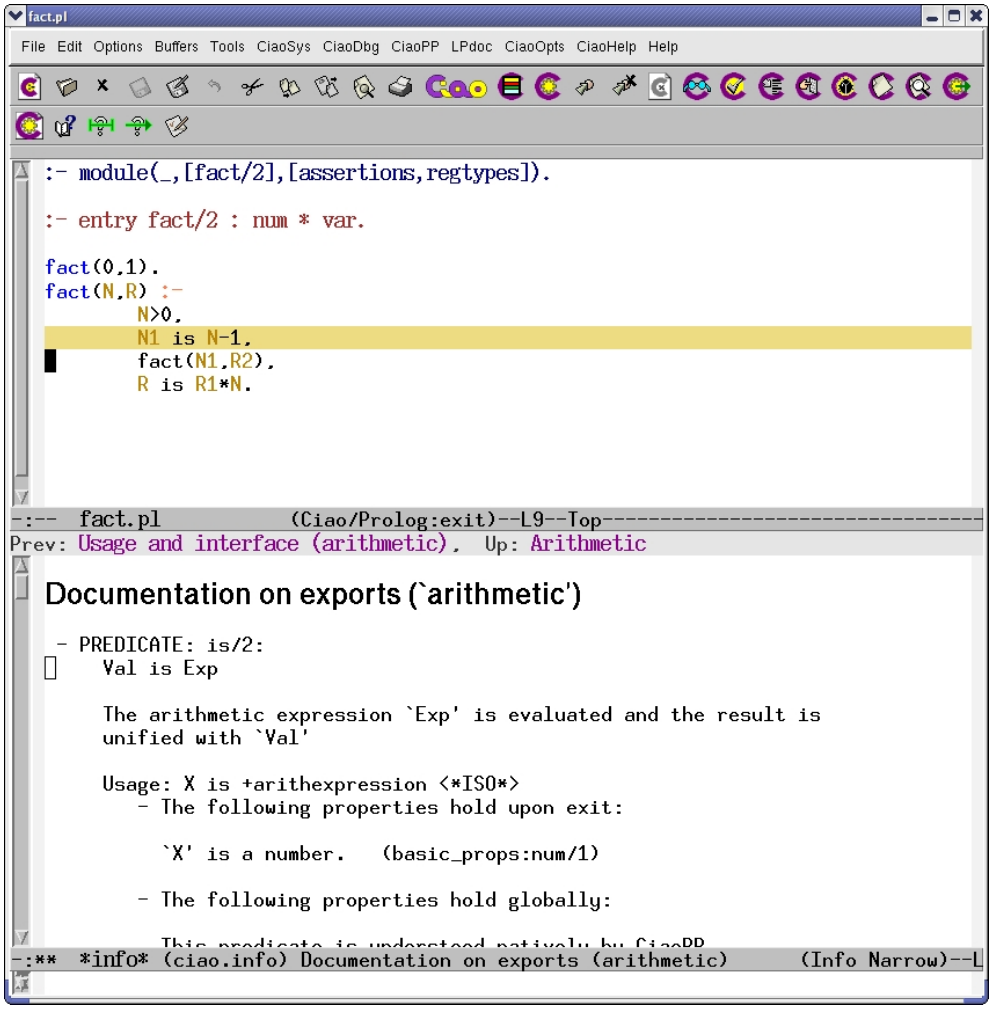
```
Customize all LPdoc environment settings
Set LPdoc executable
Set LPdoc executable args
Set LPdoc root working directory
Set LPdoc library path
```

```
Customize all Ciao colors/faces
```

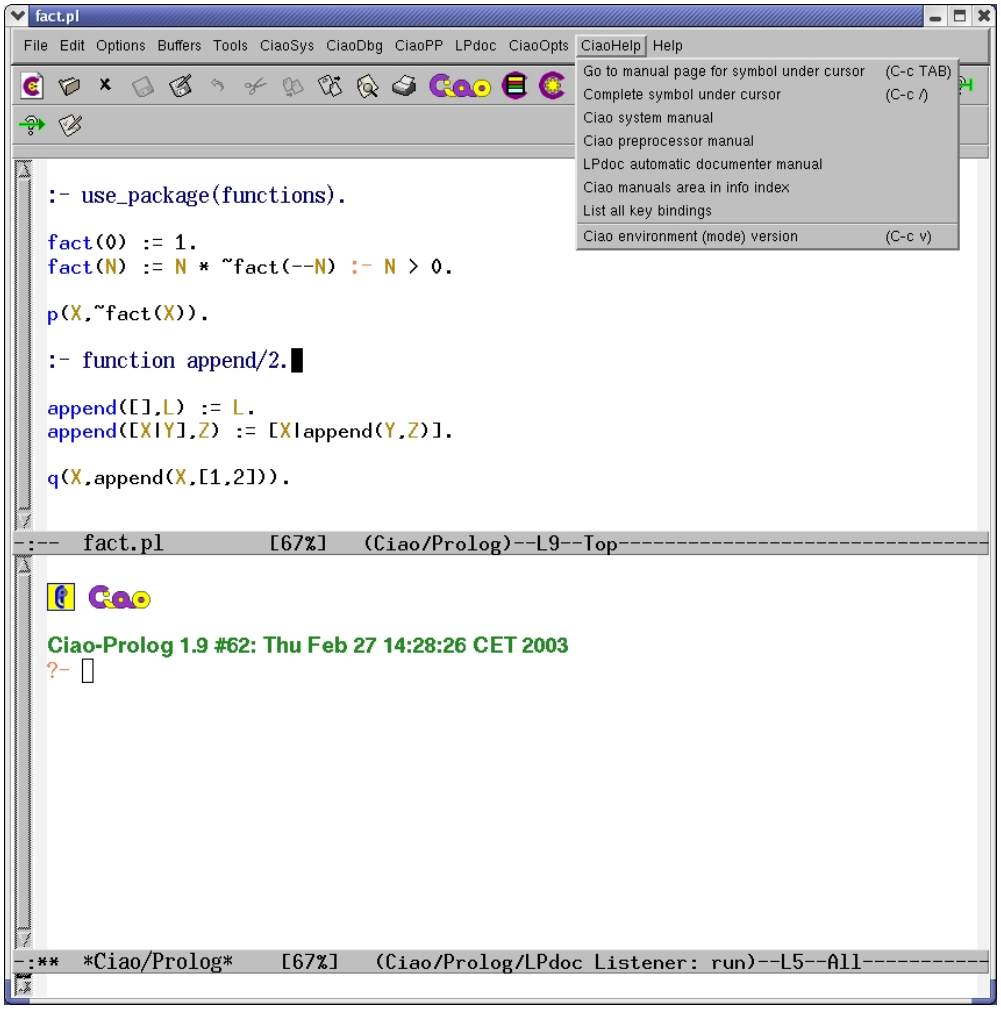
Ciao Help Menu (Partial)

```
Go to manual page for symbol under cursor  (C-c TAB)
Complete symbol under cursor               (C-c /)
Ciao system manual
Ciao preprocessor manual
LPdoc automatic documenter manual
Ciao manuals area in info index
List all key bindings


Ciao environment (mode) version            (C-c v)
```

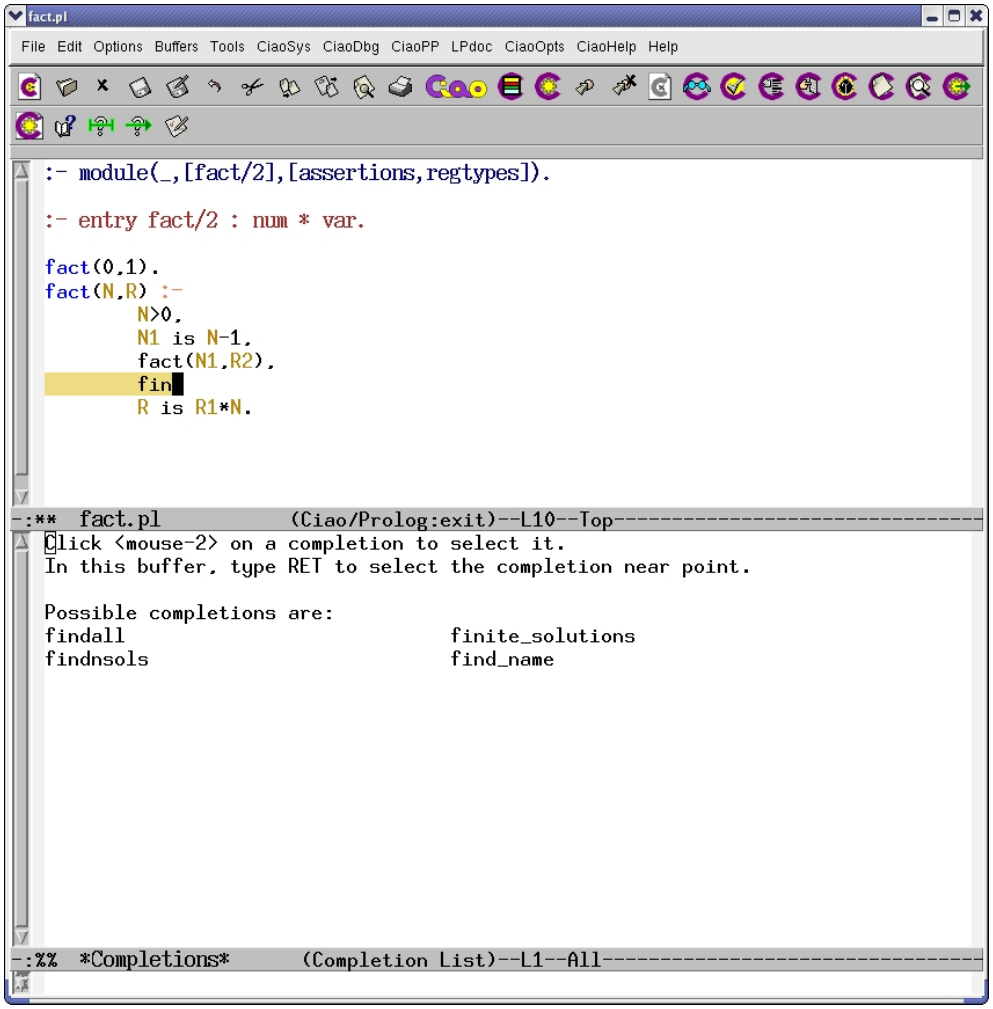
Ciao Prog. Environment: Getting Help (on predicate under cursor)



Ciao Prog. Environment: Accessing Manuals



Ciao Prog. Environment: Using Autocompletion



The Ciao Module System

The Ciao Module System

- Ciao implements a module system [12] 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:
  - `:- module(`*module_name*`, [`*exports*`], [iso]).`
    $\Rightarrow$ Iso Prolog module.
  - `:- module(`*module_name*`,[`*exports*`], [classic]).`
    $\Rightarrow$ "Classic" Prolog module
    (ISO + all other predicates that traditional Prologs offer as "built-ins").
  - Special form:
    `:- module(`*module_name*`, [`*exports*`]).`
    Equivalent to:
    `:- module(`*module_name*`, [`*exports*`], [classic]).`
    $\Rightarrow$ Provides compatibility with traditional Prolog systems.

Defining modules and exports (Contd.)

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

    If given as "_" module name taken from file name (default).
    Example: `:- module(_, [list/1, member/2]).` (file is `lists.pl`)
  - `:- 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),
     ...
```

The Ciao Module System (Contd.)

- Some more specific characteristics [12]:
  - 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).

Modular Compilation/Processing

- The Ciao compiler [13]:
  - Uses a generic program processing framework (library).
  - Can compile separately program components.
  - Builds small, standalone executables.
  - With different linking regimes.
- The actual compiler is a component used by:
  - The stand-alone compiler (`ciaoc`).
  - The top-level shell (`ciaosh`).
  - Any user executable that may need to compile programs.
- It is based on a generic program-processing *library* which:
  - Understands the module system.
  - Abstracts away many functionalities common to several modular program processing tasks.

The Generic Code-Processing Framework

- Many program processing tools (compiler, preprocessor, documenter, ...) require:
  - Reading programs into a normalized internal representation.
  - Dealing with syntactic extensions in the process.
  - "Understanding" the module system (imports, exports, multifiles, scope, etc.).
- We have abstracted this functionality into a library which offers:
  - A normalized internal representation with line numbers, etc.
  - Modular, incremental, separate, and global processing of files.
  - Dependency checking (what needs to be recompiled).
  - Automatic creation/update of interface/dependency (`.itf`) files.
  - Static detection of syntax and generic module-related errors.
  - Parameterizable via higher order.
- Used by the low-level (WAM) compiler, the preprocessor (global analyzers, etc), the automatic documentation generator, and the assertion preprocessor.
- Ensures *consistency* among the various code-processing tools.

The Ciao (Low-level) Compiler

- Makes use of the generic framework.
- Is itself also a library.
- This library is used by the standalone compiler (`ciaoc`), the top-level shell (`ciaosh`) and the Prolog script interpreter (`ciao-shell`).
- Features:
  - Global compilation/dependency-checking process.
  - Creates / maintains bytecode files (`.po`) separately for each module/user file.
  - Incrementally processes multifile-programs, (re)compiling only the files which have changed or which are affected by changes in related files.
  - Statically detects a great number of errors.
  - Builds small, standalone executables.
  - With different linking regimes.

Linking Regimes of Executables, Scripts

- The compiler produces executables by collecting the bytecode (`.po`) files of the program components, and linking them in a file to be loaded by the Ciao engine.

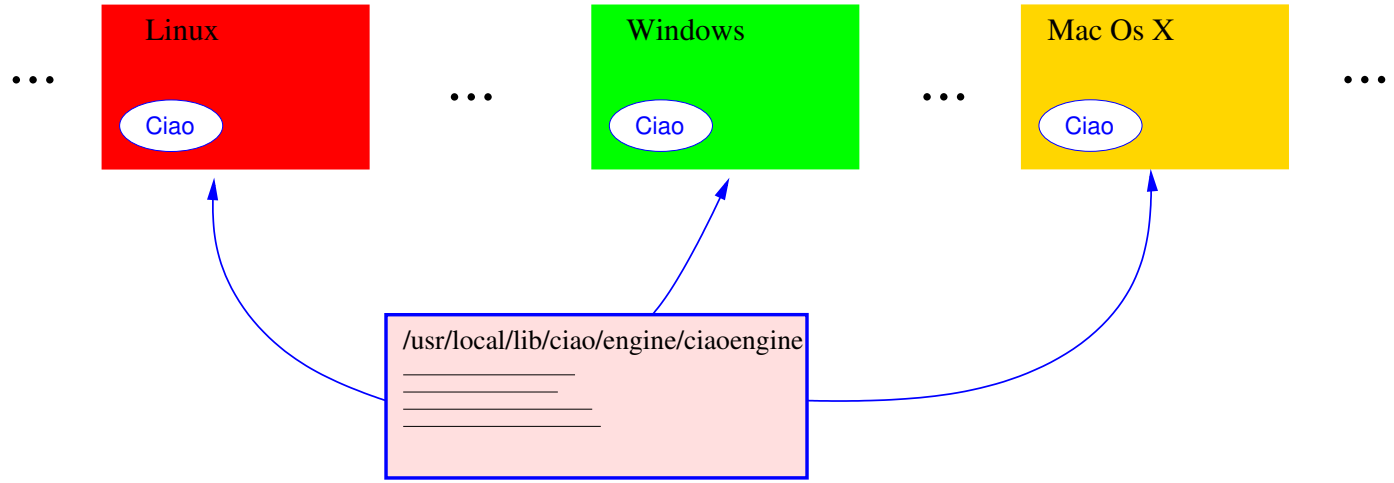
Linking Regimes of Executables, Scripts

- The compiler produces executables by collecting the bytecode (`.po`) files of the program components, and linking them in a file to be loaded by the Ciao engine.
- Modules can be linked in the executable in three ways:
  - ◇ Statically: bytecode of the module added to the executable.
    + When running the program, the module does not have to be available.
    - Larger executables and more compilation time.
  - ◇ Dynamically: bytecode loaded at startup from standard locations.
    + Smaller executable, flexibility (libraries can be updated without recompiling executables).
    - Module has to be accessible at run-time.
  - ◇ Lazily: bytecode loaded when a predicate of the module is called.
    + Useful when not all capabilities of an application are used in every run.
    - Not possible for every module. The compiler has to produce stump code.

Linking Regimes of Executables, Scripts

- The compiler produces executables by collecting the bytecode (`.po`) files of the program components, and linking them in a file to be loaded by the Ciao engine.
- Modules can be linked in the executable in three ways:
  - ◇ Statically: bytecode of the module added to the executable.
    + When running the program, the module does not have to be available.
    - Larger executables and more compilation time.
  - ◇ Dynamically: bytecode loaded at startup from standard locations.
    + Smaller executable, flexibility (libraries can be updated without recompiling executables).
    - Module has to be accessible at run-time.
  - ◇ Lazily: bytecode loaded when a predicate of the module is called.
    + Useful when not all capabilities of an application are used in every run.
    - Not possible for every module. The compiler has to produce stump code.
- Executables may be compressed: smaller but (sometimes) slower startup.
- Stand-alone architecture-dependent executables may also be created.
- *Scripts* can also be used (hide compilation/interpretation process).

Abstract Machine-based: Multiplatform

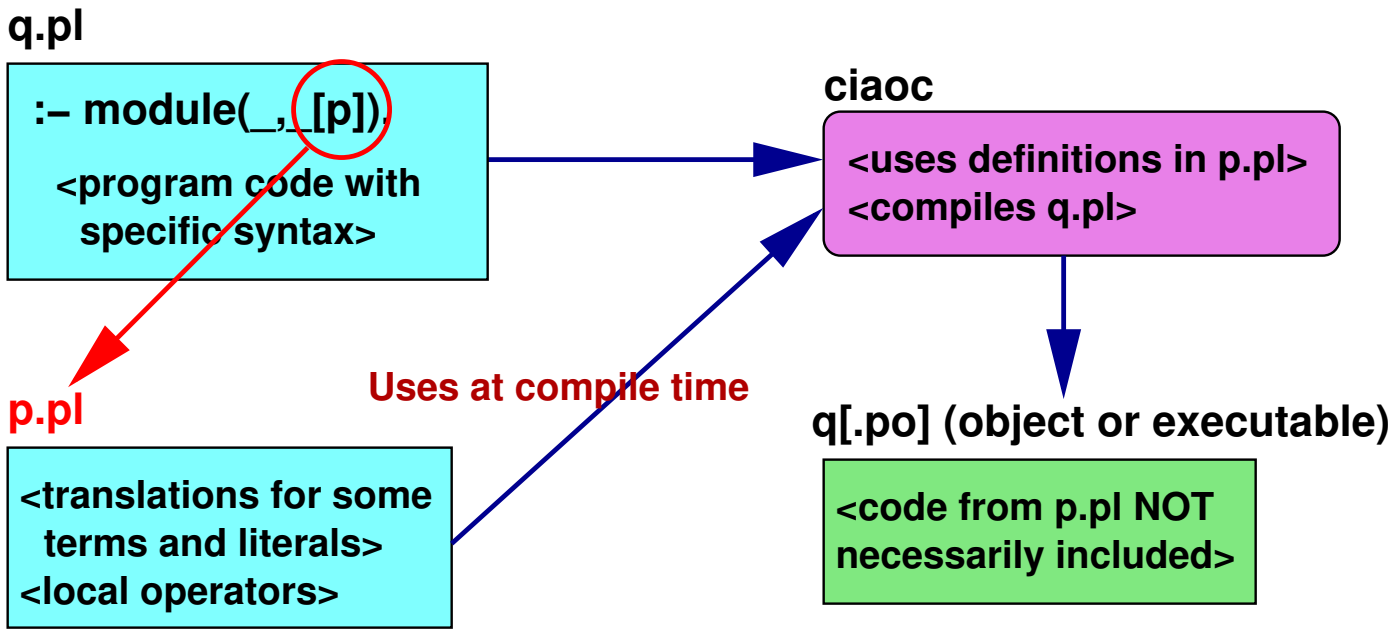


(Also for: Solaris Sparc, Solaris i386, Linux Sparc, SunOs, IRIX, ...)

Packages

- Libraries defining extensions to the language.
- Made possible thanks to:
  - ◇ Local nature of syntax extensions.
  - ◇ Clear distinction between compile-time and run-time code.
- Typically consist of:
  - ◇ A main source file to be *included* as part of the file using the library, with declarations (`op`, `new_declaration`, etc ...).
  - ◇ Code needed at compile time to make translations (loaded by a `load_compilation_module` directive).
  - ◇ Code to be used at run-time (loaded using `use_module` directives).

Packages (Cont.)



- Examples: `dcg` (definite clause grammars), `argnames` (accessing term/predicate arguments by name), `iso` (ISO-Prolog compatibility package), `functions` (functional syntax), `class` (object oriented extension), `persdb` (persistent database), `assertions` (to include program assertions – see [42]), ...

Functional Programming

Functional Notation (A Package!)

- Function applications:
  - Any term preceded by the `~/1` operator is a function application:

`write(~arg(1, T)).`	`arg(1, T, A), write(A).`

Functional Notation (A Package!)

- Function applications:
  - Any term preceded by the `~/1` operator is a function application:

```
write(~arg(1, T)).
```

```
arg(1, T, A), write(A).
```

  - Declarations can be used to avoid the need to use the `~/1` operator:

```
:- function arg/2.
```

```
write(arg(1, T)).
```

Functional Notation (A Package!)

- Function applications:
  - Any term preceded by the `~/1` operator is a function application:

```
write(~arg(1, T)).
```

```
arg(1, T, A), write(A).
```

  - Declarations can be used to avoid the need to use the `~/1` operator:

```
:- function arg/2.
```

```
write(arg(1, T)).
```

  - Also possible to use arguments other than last for "return":

```
:- fun_return functor(~,_,_).
```

```
~functor(~, f, 2).
```

Functional Notation (A Package!)

- Function applications:
  - Any term preceded by the `~/1` operator is a function application:

```
write(~arg(1, T)).
```

```
arg(1, T, A), write(A).
```

  - Declarations can be used to avoid the need to use the `~/1` operator:

```
:- function arg/2.
```

```
write(arg(1, T)).
```

  - Also possible to use arguments other than last for "return":

```
:- fun_return functor(~,_,_).
```

```
~functor(~, f, 2).
```

  - The following declaration combines the previous two:

```
:- function functor(~, _, _).
```

```
:- fun_return functor(~, _, _).
:- function functor/2.
```

Functional Notation (II)

- Several functors are *evaluable* by default:
  - Special forms for disjunctive and conditional expressions: `|/2` and `?/2`.
    - `A | B | C`
    - `Cond1 ? V1`
    - `Cond1 ? V1 | V2`

    Precedence implies that: `Cond1 ? V1 | Cond2 ? V2 | V3`
    is parsed as: `Cond1 ? V1 | (Cond2 ? V2 | V3)`

  - All the functors understood by *is/2*, if the following declaration is used:

    ```
    :- fun_eval arith(true).
    ```

    Using `false` it can be (selectively) disabled.

Functional Notation (III)

- *Functional definitions:*
  - Defining function `F/N` implies defining predicate `F/(N+1)`.

    ```
    opposite(red) := green.          ≡ opposite(red,green).
    addlast(X,L) := ~append(L,[X]).  ≡ addlast(X,L,R) :- append(L,[X],R).
    ```
  - The last argument of the predicate is assumed by default to hold the "result" of the function.
  - No run-time slow down for functions.
  - Can also have a body (serves as a guard or as *where*):

    ```
    fact(0) := 1.
    fact(N) := N * ~fact(--N) :- N > 0.
    ```
- The translation:
  - Produces *steadfast* predicates (bindings after cuts).
  - Maintains tail recursion.

Functional Notation (IV)

- The declaration `:- fun_eval defined(true).` allows dropping the ~ within a function's definition:

```
fact(0) := 1.
fact(N) := N * fact(--N) :- N > 0.
```

And, using conditional expressions:

```
fac(N) := N = 0 ? 1
        | N > 0 ? N * fac(--N).
```

Deriv and its Translation

```
der(x)      := 1.
der(C)      := 0                    :- number(C).
der(A + B)  := der(A) + der(B).
der(C * A)  := C * der(A)           :- number(C).
der(x ** N) := N * x ** ~(N - 1) :- integer(N), N > 0.
```

```
der(x, 1).
der(C, 0) :-
          number(C).
der(A + B, X + Y) :-
          der(A, X), der(B, Y).
der(C * A, C * X) :-
          number(C), der(A, X).
der(x ** N, N * x ** N1) :-
          integer(N), N > 0, N1 is N - 1.
```

Examples – Sugar for Append

- Some syntactic sugar for append:

```
:- fun_eval append/2.

mystring(X) := append("Hello",append(X,"world!")).
```

- Some more:

```
:- op(600, xfy, (.)).
:- op(650, xfy, (++)).
:- fun_eval (++)/2.
[]   ++ L := L.
X.Xs ++ L := X.(Xs ++ L).

mystring(X) := "Hello" ++ X ++ "world!".
```

Functional Notation (V)

- *Quoting.* Evaluable functors can be prevented from being evaluated:

```
pair(A,B) := ^(A-B).
```

- *Scoping.* When innermost function application is not desired (e.g., for certain meta-predicates) a different scope can be determined with the `(^^)/1` operator:

```
findall(X, (d(Y), ^^(X = ~f(Y)+1)), L).
```

translates to:
```
findall(X, (d(Y),f(Y,Z),T is Z+1,X=T), L).
```
as opposed to:
```
f(Y,Z), T is Z+1, findall(X, (d(Y),X=T), L).
```

- *Laziness.* Execution is suspended until the return value is needed:

```
:- lazy fun_eval nums_from/1.

nums_from(X) := [X | nums_from(X+1)].
```

(Can be done easily with `when`, `block`, `freeze`, etc. but proposed notation more compact for this special case. Also, `:- lazy pred_name/M`.)

Functional Notation (VI)

- Functional notation really useful, e.g., to write regular types in a compact way:

```
color := red | blue | green.
list := [] | [ _ | list].
list_of(T) := [] | [~T | list_of(T)].
```

Which translate to:

```
color(red). color(blue). color(green).

list([]).
list([_|T]) :- list(T).

list_of(_, []).
list_of(T, [X|Xs]) :- T(X), list_of(T, Xs).
```

And can then of course be used in Ciao assertions:

```
:- pred append/3 :: list * list * list.
:- pred color_value/2 :: list(color) * int.
```

Examples – Array Access Syntax

```
:- module(arrays_rt,_,[functional,hiord,assertions,regtypes,isomodes]).

:- pred fixed_array(Dim,Array) :: dim * array
   # "@var{Array} is an array of fixed dimensions @var{Dim}.".
fixed_array([N|Ms],A):- functor(A,a,N), rows(N,Ms,A).
fixed_array([N],   A):- functor(A,a,N).

rows(0,_Ms,_A).
rows(N,Ms,A):- N > 0, arg(N,A,Arg), fixed_array(Ms,Arg), rows(N-1,Ms,A).

:- regtype dim(D) # "@var{D} represents the dimensions of an array.".
dim(D) :- list(D,int).

:- regtype vector(V) # "@var{V} is a one-dimensional fixed-size array.".
vector(V) :- fixed_array([N],V), int(N).
```

- E.g., for for 2x2: `A = a(a(_,_),a(_,_)).`

Examples – Array Access Syntax (Contd.)

- We can define array access (with some syntactic sugar), also in file `arrays_rt` as follows:

```
:- include(arrays_ops).

:- pred @(Array,Index,Elem):: array * dim * int
# "@var{Elem} is the @var{Index}-th element of @var{Array}.".

V@[I]    := ~arg(I,V).
V@[I|Js] := ~arg(I,V)@Js.
```

- Where file `arrays_ops` (meant to be used by both `arrays_rt` and applications that use the syntax that we are defining, contains:

```
:- use_package(functional).
:- op(150,xfx,[@]).      :- fun_eval '@'/2.
:- op(500,yfx,<+>).      :- fun_eval '<+>'/2.
:- op(400,yfx,<*>).      :- fun_eval '<*>'/2.
```

Examples – Array Access Syntax (Contd.)

- And we can define, e.g., vector addition as:

```
:- pred <+>(V1,V2,V3) :: vector * vector * vector
# "@var{V3} is @var{V1} + @var{V2}.".
V1 <+> V2 := V3 :-
        V1 = ~fixed_array([N]), V2 = ~fixed_array([N]),
        V3 = ~fixed_array([N]), V3 = ~vecplus(N,V1,V2).

vecplus(0,_,_,_).
vecplus(N,V1,V2,V3) :- N > 0,
                       V3@[N] = V1@[N] + V2@[N],
                       vecplus(N-1,V1,V2,V3).
```

Examples – Array Access Syntax (Contd.)

- If we define an `arrays.pl` header file as follows:

```
:- include(arrays_ops).
:- use_module(arrays_rt).
```

- This new *package* then be used, e.g., as follows:

```
:- module(_,_).
:- include(arrays).

main(M) :-
        V1 = a(1,3,4,5), V2 = a(5,4,3,1), I = 1,
        display(V2@[I+1]),          % Access a given element.
        M = V1 <*> ( V2 <+> V1 ).   % Operations on vectors.
```

Combining with Constraints, etc.

- Combining with constraints, some syntactic sugar, assertions:

```
:- module(_,_,[assertions,fsyntax,clpq]).

:- fun_eval .=. /1.
:- op(700,fx,[.=.]).
:- fun_eval fact/1.

:- pred fact(+int,-int) + is_det.
:- pred fact(-int,-int) + non_det.

fact( .=. 0) := 1.
fact(N) := .=. N*fact( .=. N-1 ) :- N .>. 0.
```

- Sample query:

```
?- 24 = ~fact(X).
X = 4
```

Functional Notation (VII)

- *Definition of "real" functions:* `:- funct name/N.`

  adds pruning operators and Ciao *assertions* to add functional restrictions: determinacy, modedness, etc.

- E.g.:

```
:- funct nrev/1.
nrev( [] )    := [].
nrev( [H|T] ) := ~conc( nrev(T),[H] ).
```

Is translated to (simplified):

```
:- pred nrev(A,B,C)
      : (ground(A), ground(B), var(C))
     => (ground(A), ground(B), ground(C))
      + is_det,mut_exclusive,covered,no_fail.

nrev( [],   Y ) :- !, Y = [].
nrev( [H|L],R ) :- !, nrev(L,RL), conc(RL,[H],R).
```

Some Implementation Details

- All syntactic effects are local to the modules that use these packages (as usual in Ciao).
- Functional features provided by Ciao *packages*:
  - A first set provides the bare function features without lazy evaluation,
    - Package `fsyntax`:
      uses `fun_eval arith(false)` and `fun_eval defined(false)`.
    - Package `functional`:
      uses `fun_eval arith(true)` and `fun_eval defined(true)`. Also, `./2` as infix operator and `++/2` as infix function append are defined by default.
  - An additional one provides the lazy evaluation features.
- Functional features are implemented by translation using the well-known technique of adding a goal for each function application.

Some Implementation Details

- Translation of a lazy function into a predicate is done in two steps:
  - ◇ First, the function is converted into a predicate by the standard functions package.
  - ◇ The predicate is then transformed to *suspend its execution until the value of the output variable is needed*, by means of the `freeze/2` or `block` family of control primitives.
- (For `freeze/2` the translation will rename the original predicate to an internal name and add a *bridge predicate* with the original name which invokes the internal predicate through a call to `freeze/1`.)

Example of Lazy Functions and Translation (stylized)

```
:- lazy function fiblist/0.
fiblist := [0, 1 | ~zipWith(add, FibL, ~tail(FibL))]
        :- FibL = fiblist.
```

```
:- lazy fiblist/1.
fiblist([0, 1 | Rest]) :-
          fiblist(FibL),
          tail(FibL, T),
          zipWith(add, FibL, T, Rest).
```

```
fiblist(X) :-
          freeze(X, 'fiblist_$$lazy$$'(X)).

'fiblist_$$lazy$$'([0, 1 | Rest]) :-
          fiblist(FibL),
          tail(FibL, T),
          zipWith(add, FibL, T, Rest).
```

Higher Order

Support for Higher-Order Programming

- HO programming can be supported in Prolog via meta-programming (`=..`, `call/1`, etc.).
- Ciao provides in addition "native" HO programming (semantically cleaner), and useful syntactic extensions (e.g., the `hiord` package):
  - A family of `call/N` builtins is provided which allow the first argument of a call to `call/N` to be instantiated to:
    - A higher-order term (supporting currying), e.g.: `member(3)`
    - A "predicate abstraction": `''(X,Y) :- Y is X+10` (read `''` as λ).
  - Special syntax supported:
    `P(X,...)` is read as `call(P,X,...)`,
    `_(X,...)` is read as `''(X,...)`
  - `meta_predicate/1` declarations are extended to reflect higher-order predicates (`pred(N)`).

Simple Higher-Order Programming Examples

```
?- use_package(hiord).
?- P = <(0), P(3).
P = <(0) ?
yes

?- P = member(3), P([1,2,3]).
P = member(3) ?
yes
?- _P = member(3), _P(L).
L = [3|_] ? ;
L = [_,3|_] ?
...

?- call(member(3),L).
L = [3|_] ? ;
L = [_,3|_] ?
...
```

Simple Higher-Order Programming Examples (Contd.)

```
?- _P=member(X), _P([1,2,3]).
X = 1 ? ;
X = 2 ?
...

?- P = ( _(X,Y):- Y is X+10 ), P(2,R).
P = (''(X,Y):-Y is X+10),
R = 12 ?
yes
?- call(( _(X,Y):- Y is X+10 ), 2, R).
R = 12 ?
yes
?-
```

Extended Meta-Predicate Declarations for HO Programming

A meta-predicate specification for a predicate is the functor of that predicate applied to atoms which represent the kind of module expansion that should be done with the arguments. Possible contents are represented as:

`goal` Argument will be a term denoting a goal which will be called. For compatibility ':' can be used as well.

`pred(`*N*`)` This argument will be a predicate construct to be called by means of `call/`*N*.

`clause` This argument will be a term denoting a clause.

`fact` This argument will be a term denoting a fact.

`spec` This argument will be a predicate name (*Functor/Arity*).

`?,+,-,_` These other values denote that this argument is not module expanded.

Simple Higher-Order Programming Examples (Contd.)

- Example, parametric list regular type:

```
:- use_package([assertions,regtypes,hiord]).

:- regtype lst(T, L) # "@var{L} is a list of @var{T}s.".

:- meta_predicate lst(pred(1), ?).

lst(_, []).
lst(T, [X|Xs]) :-
        T(X),
        lst(T, Xs).
```

- Examples:
  - `lst(atom, L)` checks that a term `L` is a list of atoms.
  - `lst(lst(atom),L)` checks that a term `L` is a list of lists of atoms.
  - `lst((_(X):-write(X), nl), L)` writes all the elements of `L` (!).

Simple Higher-Order Programming Examples (Contd.)

- Example, `map/3` (see library `hiordlib`, `maplist/3`):

```
:- module(_,_,[hiord]).
:- meta_predicate map(?,pred(2),?).
map([],     _, []).
map([X|Xs], P, [Y|Ys]) :-
        P(X,Y),
        map(Xs,P,Ys).
```

- Examples of use of `map/3`:

```
?- map([[3,1,2],[c,a,b]],sort,L).
L = [[1,2,3],[a,b,c]] ?
```

```
?- map([1,3,2], (''(X,Y) :- arg(X,f(a,b,c,d),Y)), R).
R = [a,c,b] ?
```

```
?- map([X,Y],member,[[1,2],[a,b]]).
X = 1, Y = a ? ;
X = 1, Y = b ? ;
X = 2, Y = a ? ;
X = 2, Y = b ?
```

Combining Higher-Order with Functional Notation

- They can be combined, resulting in essentially functional-style code (but sometimes with additional modes of use of course!).
- Combining function application (`~`) and HO:
  - Predicate application $\Rightarrow$ Function application
    `..., P(X,Y), ...` $\Rightarrow$ `..., Y = ~P(X), ...`
- Function abstraction (only in later versions of Ciao):
  - Predicate abstraction $\Rightarrow$ Function abstraction
    `{''(X,Y) :- p(X,Z), q(Z,Y)}` $\Rightarrow$ `{''(X) := ~q(~p(X))}`

Combining Higher-Order with Functional Notation

- Some common examples:

```
:- meta_predicate map(_,pred(2),_).
map([], _) := [].
map([X|Xs], P) := [~P(X)|~map(Xs,P)].
```

```
:- meta_predicate foldl(_,_,pred(3),_).
foldl([], Seed, _Op) := Seed.
foldl([X|Xs], Seed, Op) := ~Op(X,~foldl(Xs,Seed,Op)).
```

- More uses of `map/3` (using functional notation):

```
?- L = ~map([1,2,3], ( ''(X,Y):- Y = f(X) ) ).
L = [f(1),f(2),f(3)]

?- [f(1),f(2),f(3)] = ~map(L, ( ''(X,f(X)) :- true ) ).
L = [1,2,3]

?- foldl([1,2,3],10,(''(X,Y,Z) :- Z is X+Y),R).
R = 16
```

Alternative Computation Rules

Using Other Computation (Search) 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 (`sr/bf` and `sr/bfall` packages).
  - Iterative-deepening (`id` package).
  - Depth-First search with limited depth (`id` package).
  - Fuzzy LP. Mycin.
  - "And-fair" breadth-first (`sr/af` package).
  - Tabling.
  - Andorra (deterministic-first).
- `pure` package + `sr/bf` (or `id` etc.) ideal for first steps in teaching LP!

Breadth-First

- Use package `bf` (`sr/bf`).
- Predicates written with the operator '`<-`' are executed using breadth-first search.
- Normal predicates and breadth-first predicates can be freely mixed in the same module.
- The `sr/bfall` package makes *all* predicates in a module be executed breadth-first (in this case it is necessary to write rules and facts using `<-`, i.e., standard syntax can be used.
- The `sr/af` version ensures "and-fairness" by goal shuffling.

Breadth-First Example I

```
:- module(chain, [test/1], [sr/bf]).

test(df) :- chain(a,d).     % Loops with usual depth first rule
test(bf) :- bfchain(a,d).

bfchain(X,X) <- .
bfchain(X,Y) <- arc(X,Z), bfchain(Z,Y).

chain(X,X).
chain(X,Y):- arc(X,Z), chain(Z,Y).

arc(a,b).   arc(a,d).
arc(b,c).   arc(c,a).
```

Breadth-First Example II

```
:- module(chain_bfall, _, [sr/bfall]).

test :- chain(a,d).

chain(X,X).
chain(X,Y) :- arc(X,Z), chain(Z,Y).

arc(a,b).
arc(a,d).
arc(b,c).
arc(c,a).
```

Breadth-First Example III

```
:- module(sublist, [test/1], [sr/af]).

test(df) :- sublist_df([a],[b]).  % loops with depth first rule.
test(bf) :- sublist_bf([a],[b]).  % loops with normal breadth-first

sublist_df(S,L) :- append(_,S,Y), append(Y,_,L).

sublist_bf(S,L) <- append(_,S,Y), append(Y,_,L).

append([], L, L) <- .
append([X|Xs], L, [X|Ys]) <- append(Xs, L, Ys).
```

Iterative-Deepening

- Modules can be marked to have iterative deepening behavior.
- A directive sets the initial depth, the predicate that computes the increment, and, optionally, a maximum depth.

  Examples:

```
:- iterative(p/1,5,f).   % to start with depth 5 and increment by 10
f(X,Y) :- Y is X + 10.
                         % or, using predicate abstactions
:- iterative(p/1,5,(_(X,Y):- Y is X + 10)).

:- iterative(p/1,5,(_(X,Y):- Y is X + 10),100). %  All goals below
                                                %  100 simply fail
```

- Bounded depth-first can be done by one-step iterative-deepening:

```
:- iterative(p/1,100,f,100).
```

Constraint Programming

Constraints

- Current packages: `clpq` and `clpr`.
  Based on Holzbaur's implementation [33, 32] using attributed variables.
- (Also limited support for finite domains –`fd` package.)
- The effect of loading `clpq` or `clpr` is local to a module.
- CLP($\mathcal{Q}$) is exact, CLP($\mathcal{R}$) is (obviously) approximate.
- Constraints must be written using special operators: `X .=. Y+Z`, `X .=<. 2*Y`
- Linear equations are checked for satisfiability immediately, nonlinear equations are delayed until they become linear.
- The packages are also usable directly in the toplevel:

```
?- use_package(clpq).
{ some messages }
?- X*Y .>. Z, X+2*Y .=. 10, X .=. Y/3.
X = 10/7, Y = 30/7, Z.<.300/49 ?
```

- Other constraint domains (e.g., finite domains) in development.

CLP example

- Fibonacci relation:

```
fib(0, 0).
fib(1, 1).
fib(N, F) :-
        N .>. 0,
        N1 .=. N - 1,
        N2 .=. N - 2,
        F .=. F1 + F2,
        fib(N1, F1),
        fib(N2, F2).
```

CLP example

- Fibonacci relation:

```
fib(0, 0).
fib(1, 1).
fib(N, F) :-
        N .>. 0,
        N1 .=. N - 1,
        N2 .=. N - 2,
        F .=. F1 + F2,
        fib(N1, F1),
        fib(N2, F2).
```

- Finding fixpoints:

```
?- N = ~fib(N).
N = 0 ? ;
N = 1 ? ;
N = 5 ?
```

Another CLP example

- Example: placing N queens in a N*N board

```
queens(N, Qs) :- constrain_values(N, N, Qs), place_queens(N, Qs).

constrain_values(0, _N, []).
constrain_values(N, Range, [X|Xs]) :-
        N .>. 0,
        X .>. 0, X .=<. Range,
        N1 .=. N - 1,
        constrain_values(N1, Range, Xs), no_attack(Xs, X, 1).

no_attack([], _Queen, _Nb).
no_attack([Y|Ys], Queen, Nb) :-
        Queen .<>. Y,       % this line missing in the slides!!
        Queen .<>. Y+Nb,
        Queen .<>. Y-Nb,
        Nb1 .=. Nb + 1,
        no_attack(Ys, Queen, Nb1).
```

Object-Oriented Programming

Object-Oriented Features: O'Ciao

- Basic design philosophy [40]:
  - ◇ Identify desired feature(s) of OOP not present or difficult to use in LP/Ciao.
  - ◇ Add them in the most natural way.
  - ◇ Blend object model as much as possible with existing LP/Ciao concepts and features (e.g., the module system).

Feature	OOP	Correspondence in Ciao
State	attributes	dynamic predicates
Encapsulation	classes	modules
Polymorphism	overloading	clause selection
Instantiation	objects	–
Inheritance	classes	(reexport)

- Missing is instantiation $\rightarrow$ use module system / dynamic predicates; add:
  - ◇ Module instantiation.
  - ◇ Other features (virtual methods, interfaces, inheritance, ...).

Ciao Instantiable Modules → Classes/Objects

- `new/2`: conceptually creates a dynamic "copy" of a module.
  (But implemented more efficiently!)
- Effectively, implements a very useful notion of classes/objects.
- Example:

```
:- class(deck,[addcard/1,drawcard/1]).

:- dynamic card/2.
% initial state
card(1,hearts).
card(8,diamonds).

addcard(card(X,Y))  :- asserta(card(X,Y)).
drawcard(card(X,Y)) :- retract(card(X,Y)).
```

```
:- module(main,[main/0],[objec
:- use_class(deck).

main :-
     S1 new deck,
     S2 new deck,
     S1:drawcard(C),
     S2:addcard(C).
```

Ciao Instantiable Modules → Classes/Objects (Contd.)

- Same calling syntax as for the module system.
- Visibility controlled by the same rules as in the module system.
- Object state is represented by the state of the dynamic predicates.

- Similar capabilities to other designs (e.g., SICStus objects, Logtalk, ...).
  But those are typically unrelated to the module structure.
- O'Ciao adds a number of features:
  - Inheritance (based on the module system reexport capabilities + syntactic sugar).
  - Overriding: just export new predicate with other name.
  - Abstract methods (e.g., `virtual` declarations).

O'Ciao: Defining Classes

- The `module` declaration is replaced by a `class` declaration.

  `:- class(stack).` $\equiv$ `:- module(stack,[],[class]).`

- Predicates are interpreted as (instance) methods.
- The usual `export` declarations define the public interface of the class: the visible methods for the class instances.

  ```
  :- export(push/1).
  :- export(pop/1).
  ```

  or

  ```
  :- class(stack,[push/1,pop/1]).
  ```

- The `dynamic` and `data` declarations are the attribute declarations.

  ```
  :- dynamic storage/1.
  ```

  or

  ```
  :- data storage/1.
  ```

- Attributes are easily initialized by writing facts for them.

Another Example of a Class

```
:- class(stack,[push/1,pop/1,top/1,is_empty/0]).

% Attribute
:- data storage/1.

% Methods
push(Item) :- asserta_fact(storage(Item)).

pop(Item) :- retract_fact(storage(Item)).

top(Top) :- storage(Top), !.

is_empty :- \+ storage(_).
```

O'Ciao: Using Classes

- The `objects` package enables creating and using objects from imported classes:

```
:- use_package(objects).
```

- The `use_module` declaration is replaced by a `use_class` declaration.

```
:- use_class(stack).
```

- The `new` operator enabled by the `objects` package allows instance creation.

```
...:- ..., X new stack, ...
```

- Object identified by "instance qualification"
  (resembling module qualification)

```
..., X:push(Item), ...
```

O'Ciao: Other Features

- Inheritance:
  - Obtained via extension of the reexport capabilities of the module system.
  - Some syntactic sugar provided (`inheritable/1`, `inherit_class/1`).
- Overriding:
  - Inherited methods *overridden* by new predicate declaration for them in the subclass.
  - `self/1`.
  - Follows also module system conventions.
- Abstract methods (`virtual` declarations), refinement.
- *Interfaces* used to simulate multiple inheritance (as in Java).

Other Extensions

Records package (named arguments)

- Provides named access to arguments of terms and literals.
- Example:

```
:- use_package(argnames).
:- argnames person(name, age, profession).
p(person${}).
q(person${age=> 25}).
r(person${name=> D, profession=>prof(D),age=>age(D)}).
s(person${age=>t(25), name=> daniel}).
```

Translates to:

```
p(person(_,_,_)).
q(person(_,25,_)).
r(person(A,age(A),prof(A))).
s(person(daniel,t(25),_)).
```

- Can add an argument to a predicate globally by simply adding in `:- argnames`!

Persistent Predicates

- Persistent Predicate [21, 7]: dynamic predicate residing in non-volatile media.
- Its state survives across successive executions of the application.
- Usage transparent to the storage media, and similar to normal data (dynamic) predicates.
- Changes to the persistent predicates are recorded atomically and transactionally:
  - Security against possible data loss due to, for example, a system crash.
  - Allows concurrent updates from different programs.
- Update primitives similar to `assert/1` and `retract/1`.
- Transactional behavior.
- Currently supported storage media:
  - Files: `persdb` package.
  - SQL database: `persdb_sql` package.

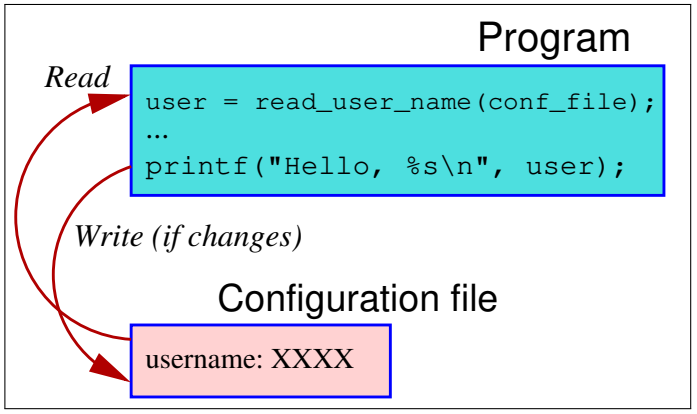
Persistent Predicates Example I

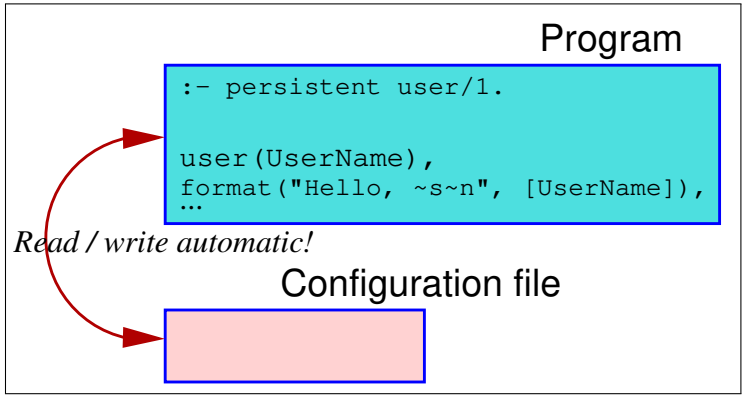
- Example: persistent queue.

```
persistent_dir(queue_dir,'./DB').
:- persistent(queue/1, queue_dir).

main:- write('Action ( in(Term). | out. | halt. ): '),
       read(A),
       ( handle_action(A) -> true ; write('Unknown command.'), nl ),
       main.

handle_action(halt) :- halt.
handle_action(in(Term)) :- passertz_fact(queue(Term)), main.
handle_action(out) :-
     (  pretract_fact(queue(Term))
     -> write('Out '), write(Term)
     ;  write('FIFO empty.') ),
     nl, main.
```

Persistent Predicates Example II

- Example: program configuration files, which need to be read and written for every run (e.g., `~/.XXXrc` files).
- Not a difficult problem, but certainly a hassle.
- Normally manual handling (read/parse/write) of the file.





- Alternitive: use persistent facts.
- State that the facts will *live* in, e.g., `~/.XXXrc`.
- Access is automatic and update as simple as using assert and retract.

Concurrency and Distribution

Basic Concurrency

- (Low-level) Concurrency in Ciao is currently provided [15] by two sets of primitives:
  - Primitives to spawn and control independent execution threads.
  - Primitives to synchronize and share information among threads.
- Spawning-related primitives provide basic control on threads.
- Threads are flat: they offer a basic mechanism on top of which more involved formalisms (e.g., concurrent objects) are built.
- Communication/synchronization implemented through accesses to the shared database:
  - Predicates declared `concurrent` have a special regime access: calls suspend instead of failing if no matching clause exists at the time of the call.
  - Backtracking can take place after suspension.
  - All accesses and updates are atomic.
  - Other primitives can change the behavior of concurrent predicates at runtime.

A Simple Example

- Start several predicates which wait for a fact to appear.

```
:- concurrent proceed/1.

waitf:-
     retract_fact(proceed(X)),
     display(proceeding(X)),
     nl.
```

```
wait_facts:-
     eng_call(waitf, create, create),
     eng_call(waitf, create, create),
     eng_call(waitf, create, create),
     asserta_fact(proceed(1)),
     asserta_fact(proceed(2)),
     asserta_fact(proceed(3)).
```

- The `concurrent/1` directive instructs the compiler to mark `proceed/1` as a concurrent predicate: calls will *suspend* if needed.
- `wait_facts/0` starts three threads in **separate** stack sets (`create` parameter).
- Each of them will atomically wait for and retract a clause of the predicate.
- Threads are executed **in parallel** when using a multiprocessor machine.

A Threaded TCP/IP-based Server

- Will wait for a connection, read two numbers, add them, and return the result (several `handle_conn/0` have been previously started).

```
:- concurrent conn/1.

wait_for_queries(Socket):-
     repeat,
     socket_accept(Socket, Stream),
     assertz_fact(conn(Stream)),
     fail.
```

```
handle_conn:-
     retract_fact(conn(Stream)),
     read(Stream, Number1),
     read(Stream, Number2),
     Result is Number1 + Number2,
     write(Stream, Result),
     close(Stream),
     fail.
```

- The main loop listens on a port and asserts stream ids as connections arrive.
- Each `handle_conn/0` waits for a `conn/1` to appear; it gets a `Stream` from which numbers to add are read.
- It fails after the answer is returned $\longrightarrow$ goes back to waiting for a new `conn/1`!

Higher-Level Concurrency Primitives

- `eng_call/3` (and others to, e.g., perform backtracking on concurrent goals) are often too low level.
- Better primitives are built on top of them (see [22, 10] for other primitives):
  - `Goal &> Handle` executes `Goal` in a separate environment, leaves a `Handle` pointing to the computation.
  - `Handle <&` waits (if necessary) for the end of the computation and installs the bindings locally.
  - Communication transparently implemented through the shared database.
- A library implementing these operators allows the programmer to write concurrent code with *arbitrary explicit* dependencies:

```
concurr:-  a &> Ha, b &> Hb, c &> Hc,
           ..., Hb <&,  ..., Ha <&, ..., Hc <&, ...
```

Higher-Level Concurrency Primitives (Contd.)

- A library implementing them allows writing concurrent code such as:

```
sort([X|Xs], Sorted):-
     partition(Xs, X, Big, Small),
     sort(Big,B) &> H,
     sort(Small,S),
     H <&, append(S, [X|B], Sorted).
```

- And also, as a particular case, the library implements the good, old fork and join of &-Prolog:

```
sort([X|Xs], Sorted):-
     partition(Xs, X, Big, Small),
     sort(Big,B) &
     sort(Small,S),
     append(S, [X|B], Sorted).
```

Distributed Execution

- It is very easy for example to write a server which listens on a port for goals and executes them [10] (recall the TCP/IP-based server):

```
:- concurrent conn/1.

wait_for_queries(Socket):-
     repeat,
     socket_accept(Socket, Stream),
     assertz_fact(conn(Stream)),
     fail.
```

```
handle_query:-
     retract_fact(conn(Stream)),
     read(Stream, Goal),
     call(Goal),
     write(Stream, Goal),
     close(Stream),
     fail.
```

- This is a simple implementation of a *goal server* for distributed execution:
  - ◇ Clients connect to a server and send goals, keeping a local handle.
  - ◇ The server reads and executes the goals.
  - ◇ Clients, when needed, ask for answers.
  - ◇ Bindings are sent back with each answer, and are locally installed using logical variables stored in the handle.

Distributed Execution (Contd.)

- A more complete server implies a more complex negotiation: a unique identifier per remote goal, remote backtracking, remote cut...
- We use the syntax of [10] (extension of the high-level concurrency primitives).
- From the client point of view:

```
p(...):- ..., r(X) @ Host > Handle, ..., Handle <&, ...
```

- `Handle` encapsulates:
  - The initial `Goal` (including logical variables),
  - the `Host` we want to execute our work,
  - a unique identifier for the communication.
- Extensions: active objects, code, and computation mobility:
  - Creation and remote invocation: `Obj new class @ Host, Obj:method(Arg).`
  - Also mobility: `Obj @ NewHost`.
  - Ongoing work [16].

Active Modules / Active Objects

- Modules to which computational resources are attached.
- High-level model of client-server interaction.
- An active module is a network-wide server for the predicates it exports.
- Any module or application can be converted into an “active module” (active object) by compiling it in a special way (creates an executable with a top-level listener).
- Procedures can be imported from remote “active modules” via a simple declaration: E.g. `:- use_active_module(Name, [`P_1/N_1`, `P_2/N_2`,...]).`
- Calls to such imported procedures are executed remotely in a transparent way.
- Typical application: client-server. Client imports module which exports the functionality provided by server. Access is transparent from then on.
- Built as an abstraction on top of ports/sockets
  (also a free library for SICStus and other systems).

Using Active Modules: An Example

- Server code (active module), file `database.pl`:

```
:- module(database, [stock/2]).

stock(p1, 23).
stock(p2, 45).
stock(p3, 12).
```

- Compilation: "`ciaoc -a` *`address publishing method`* `database`" or:

```
?- make_actmod('/home/clip/public_html/demo/pillow/database.pl',
               'actmods/filebased_publish').
```

  produces executable called `database`.

- Active module started as a process – e.g., in Unix:
  `database &`

Using Active Modules: An Example

- Client (file `sales.pl`):

```
:- module(sales,[need_to_order/1],[actmods]).
:- use_active_module(database, [stock/2]).
:- use_module(library('actmods/filebased_locate')).

need_to_order(P) :-
        stock(P, S),
        S < 20.
```

- Usage:

```
?- use_module(sales).
?- need_to_order(X).
```

Active Objects, Code, and Computation Mobility

- Code mobility is easy: code just a set of terms or string of bytecode.
- Migrating active computations is heavy from an implementation point of view: need to stop the engine, save state, reinstall O.S.-dependent data structures...
- Easy in continuation-based systems as BinProlog (but they have other problems).
- *Migrating objects* makes sense: they have local state.
- State of *Ciao* objects: set of facts $\longrightarrow$ set of terms.
- Objects can be transparently put in a *blocked* state by the *object server*: do not accept new invocations, keep track of the finished operations.
- Moving objects can performed by:
  - *Blocking* the object.
  - Sending static code (if needed) to target host + its state (dynamic code).
  - Notifying those which may want to connect to the object the new location.
- Several algorithms possible for the last point: work in progress in this area.

Web Programming

Web Programming

- The PiLLoW library simplifies the process of writing Internet and WWW applications [9, 11, 8].
- Provides facilities for:
  - ◇ Generating HTML/XML structured documents by handling them as Herbrand terms (bidirectional syntax conversion).
  - ◇ Writing CGI executables.
  - ◇ Producing HTML *forms*.
  - ◇ Writing form handlers: form data parsing.
  - ◇ Accessing and parsing WWW documents.
  - ◇ Using HTML templates.
  - ◇ Handling cookies.
  - ◇ Accessing code posted at HTTP addresses.
- See specific tutorial on the PiLLoW system.

Form Producer/Handler Example

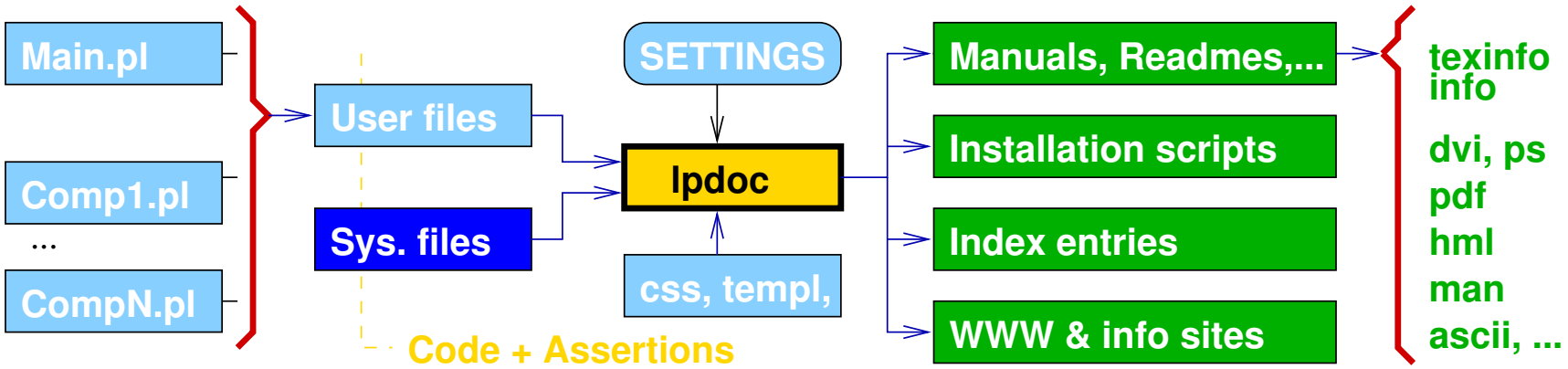
```
main(_) :-
    get_form_input(Input),
    get_form_value(Input,person_name,Name),
    response(Name,Response),
    file_to_string('html_template.html', Contents),
    html_template(Contents, HTML_terms, [response = Response]),
    output_html([cgi_reply|HTML_terms]).

response(Name, []) :- form_empty_value(Name), !.
response(Name, ['Phone number for ',b(Name),' is ',Info, --]) :-
        phone(Name,Info), !.
response(Name, ['No phone number available for ',b(Name), '.', --]).

%% Database
phone('Hanna', '613 460 069').
(...)
```

Support for Auto-Documentation (LPdoc)

LPdoc: the Ciao Automatic Documentation System

- Writing and, specially, maintaining program documentation is hard
  $\rightarrow$ automate process as much as possible.
- Objectives:
  - Keep documentation close to source
    (easy to keep in sync with the program – "Literate Programming").
  - Be able to *reuse* typical program documentation.
  - Integrate closely with assertion language used in debugging/verification.
  - Produce useful documentation even if no comments or assertions in program.
  - Integrate in program development environment (e.g., version control system).
  - Allow complex manuals (indices, images, citations from BiBTeX dbs, etc.).
  - Support many output formats.
  - Perform several related tasks (e.g., construction of distribution sites).
  - Allow text reuse in multiple places (e.g., manuals, readmes, distribution sites, lists of manuals and sw packages, announcements, installation scripts, ...)
  - Be largely (CLP) platform-independent and modular.

LPdoc Overall operation



- User view:
  - Creating manual:
    - Edit `SETTINGS` file
    - `lpdoc` *format* (`dvi`, `ps`, `html`, ...)
  - Viewing manual: `lpdoc dviview`, `lpdoc htmlview`, ...
  - Installing manual: `lpdoc install`
  - + cleanup, etc.

LPdoc Inputs

- Basic types of input files:
  - Files to be documented (possibly including assertions and comments).
  - Used but not documented (library) files
    (e.g., system and user libraries: types, properties, reexports, etc.).
  - `SETTINGS`, template files, HTML style (css files), etc.
- `SETTINGS`:
  - Determines main file and components.
  - Defines the paths to be used to find files (independent of the paths used by the `LPdoc` application itself).
  - Selects indices (predicates, ops, declarations, properties, types, libraries, concepts, authors, ...), options, etc.
  - Selects location of BiBTeX file(s), HTML styles, etc.
  - Defines installation location, etc.

Assertions

- Assertions:
  - Written in the Ciao assertion language [42].
  - Declarations, used to:
    - state general properties, types, modes, exceptions, ...
    - of certain program points, predicate usages, ...
  - Includes standard compiler directives (`dynamic`, `meta_predicate`, etc.).
  - Have a certain qualifier: check, true, trust, ...
  - Can include documentation text strings.

```
:- pred sort(X,Y)
      : list(X) => sorted(Y)
      # "@var{Y} is a sorted permutation of @var{X}.".
```

- Natively understood by `LPdoc` [23] to generate documentation (and by CiaoPP[25]).

Assertions (Contd.)

- Examples – pred:

```
:- pred qsort(X,Y) : list(X) => sorted(Y)
                   # "@var{Y} is a sorted permutation of @var{X}.".
```

- Examples – prop, regtype:

```
:- prop sorted(X) # "@var{X} is sorted.".
sorted([]).
sorted([_]).
sorted([X,Y|R]) :- X < Y, sorted([Y|R]).

:- regtype list(X) # "@var{X} is a list.".
list([]).
list([_|T]) :- list(T).
```

Comments

- Declarations, typically containing textual comments (`:- comment` (old) or `:- doc`):
  `:- comment(`*`CommentType`*`,`*`CommentData`*`).`
- Examples:

```
:- doc(title,"Complex numbers library").
:- doc(summary,"Provides an ADT for complex numbers.").
:- doc(ctimes(X,Y,Z),"@var{Z} is @var{Y} times @var{X}.").
```

- Markup language, close to LaTeX/texinfo:
  - Syntax: `@`*command* (followed by either a space or {}), or `@`*command*{*body*}.
  - Command set kept small and somewhat generic, to be able to generate documentation in a variety of formats.
  - Names typically the same as in LaTeX.
  - Types of commands:
    - Indexing and referencing commands.
    - Formatting commands.
    - Inclusion commands, etc.

Structure of generated documents

- Overall structure:
  - Single file $\rightarrow$ simple manual without chapters.
  - Multiple files:
    - Main file gives title, author(s), version, summary, intro, etc.
    - Other ("component") files are chapters and appendices.
- Chapters:
  - If file does not define `main` $\rightarrow$ assumed *library*, *interface* (API) documented.
    else $\rightarrow$ assumed *application*, *usage* documented.
  - Structure:
    - Chapter title/subtitle (or file name if unavailable).
    - Info on authors, version, copyright, ...
    - Chapter intro.
    - Interface (usage, exports, reexports, decls, ops, modules used, ...).
    - Documentation for decls, preds, props, regtypes, multifiles, modedefs,...
    - Bugs, changelog, appendices, ...

Documentation of predicates, props, etc.

- If no declarations or comments:
  - ◇ One line stating predicate name and arity
    (useful: goes to index → automatic location, automatic completion).
  - ◇ If property or regtype: source code (often best description).
- Comments for the predicate/property/regtype...
- All assertions, described in textual form (unless stated otherwise).
- `pred` assertions documented as “usages”.
- Comments associated with `pred` assertions used to describe the usages.
- Syntactic sugar can be kept or expanded.
- The text in properties is *reflected* into the predicates which use such properties (also if property is imported from another module).

The Ciao Assertion Language

Introduction

- Assertions have multiple uses/roles traditionally:
  - ◇ Run-time checking (e.g., pre/post-cond) – general properties, *"check"*.
  - ◇ Compile-time checking (e.g., types) – decidable, compulsory, *"check"*.
  - ◇ Replacing oracles in, e.g., decl. debug. – general (decl.) properties, *"check"*.
  - ◇ Providing info to an optimizer (e.g., pragmas) – general properties, *"trust"*.
  - ◇ General comm. w/compiler (e.g., entry) – general properties, *"trust"*.

- Important issue: whole system should deal *safely* with general, undecidable properties, and incomplete information $\rightarrow$ *safe approximations*.
- Assertion language proposed [42] suitable for *all* these purposes.
  (When possible, keeps backwards compatibility w/ISO & popular platforms.)
- Different program development tools may use different parts of the language.

Assertions

- Written by the user or by program processing tools.
- If written by the user they:
  - Describe the intended (declarative or operational) semantics ($\mathcal{I}$).
  - Can also be used to guide the analyses, state convenient specializations, declare the behaviour of external procedures, etc.
- In general they are *optional*.
- State *properties* (see later) of:
  - call points to procedures (*preconditions*),
  - success points (*postconditions*),
  - whole executions,
  - intermediate program points, etc.
- Apply to *all* run-time invocations of a predicate, in the current context (i.e., in the module the predicate is in, with its declared entry points –exports, etc.).

Properties

- Arbitrary predicates, (generally) *written in the source (logic) language*.
- But some conditions on them: termination, no instantiation, ...
- Some predefined in system libs, some of them “native” to an analyzer.
- Others user-defined.
- Should be visible in the module and “runnable:” they will be used also as run-time tests! (but the property may be an approximation itself).
- *Types* are a special case of property (e.g., `regtypes`).
- But also, e.g., argument sizes, instantiation states, ...

```
:- regtype list/1.
list([]).
list([_|Y]) :- list(Y).
```

```
:- prop sorted/1.
sorted([]).
sorted([_]).
sorted([X,Y|Z]) :- X>Y, sorted([Y|Z]).
```

```
:- regtype list/1.
list := [] | [_|~list].
```

```
:- regtype int/1 + impl_defined.
```

```
:- regtype peano_int/1.
peano_int(0).
peano_int(s(X)) :- peano_int(X).
```

Basic Predicate Assertions: *Success*

- *"Success"* assertions:

  `:- success` $PredPattern$ `=>` $PostCond$`.`

  - Describe *post-conditions* of a predicate.
  - $PostCond$ is a conjunction of properties.
  - Example:

    ```
    :- success qsort(A,B) => ground(B).
    ```

- Restricting to a subset of calls:

  `:- success` $PredPattern$ `:` $PreCond$ `=>` $PostCond$`.`

  - $PreCond$ is a conjunction of properties.
  - Examples:

    ```
    :- success qsort(A,B) : list(A) => list(B).
    ```

    ```
    :- success qsort(A,B) : (list(A),ground(A)) => (list(B),ground(B)).
    ```

- Success assertions never impose conditions on how predicates are called: they only state the sucess state for some or all calls to the predicate (if they succeed).

Basic Predicate Assertions: *Calls*

- *"Calls"* assertions:

  `:- calls` $PredPattern$ `:` $Props$`.`

  - Describe properties of the calls to a predicate.
  - $Props$ is a conjunction of properties.
  - Calls are closed i.e., the set of calls assertions covers *all* calls to a predicate that can occur in the environment (module, entries) in which the predicate appears.
  - Example:

```
:- calls qsort(A,B) : (list(A),var(B), indep(A,B)).
```

Basic Predicate Assertions: *Comp*

- *"Comp"* assertions:

  `:- comp` $PredPattern$ `:` $PreCond$ `+` $CompProps$`.`

  `:- comp` $PredPattern$ `+` $CompProps$`.`

  - Describe props of the whole exection of the predicate.
  - $CompProps$ is a conjunction of computational properties (determinacy, non-failure, cost, ...).
  - Example:

```
:- comp qsort(A,B) : (list(A,int),var(B)) + (is_det,not_fails).
```

- Most general, but others always preferred if possible.

Compound Predicate Assertions: *Pred* Assertions

- Issues in practice with previous assertions:
  - ◇ Verbose in some cases: more compact notation desired.
  - ◇ When writing multiple success assertions one often wants to also say that this covers all calls (this needs an additional calls assertion).
- *"Pred"* assertions are convenient "macros" for this:

  `:- pred` *PredPattern* `[ :` *Pre* `] [ =>` *Post* `] [ +` *Comp*`].`

  (Fields in `[...]` are optional, but at least one must be present.)
  - ◇ *Closed* on calls: cover all uses of a predicate (they imply a calls assertion).
  - ◇ Several form a conjunction (if several match $\rightarrow$ then GLB).
- Some examples:
  - ◇
    ```
    :- pred qsort(X,Y) => sorted(Y).
    ```
  - ◇
    ```
    :- pred qsort(X,Y) : (list(X,int),var(Y)) => sorted(Y) + (is_det,not_
    :- pred qsort(X,Y) : (var(X),list(Y,int)) => ground(X) + not_fails.
    ```
  - ◇
    ```
    :- pred foo(X,Y) : (ground(X),var(Y)) => (ground(Y),X>Y) + det.
    :- pred foo(X,Y) : (var(X),ground(Y)) => (ground(X),X>Y).
    ```

Example of a Program with Assertions

```
:- module(qsort,[qsort/2],[assertions,regtypes]).

:- pred qsort(A,B) : list(A) => sorted(B).
qsort([],[]).
qsort([X|L],R) :-
        partition(L,X,L1,L2),
        qsort(L2,R2), qsort(L1,R1),
        append(R1,[X|R2],R).

:- pred partition(A,B,C,D) : list(A).
partition([],B,[],[]).
partition([E|R],C,[E|Left1],Right):- E < C, !,
        partition(R,C,Left1,Right).
partition([E|R],C,Left,[E|Right1]):- E >= C,
        partition(R,C,Left,Right1).
```

Example of a Program with Assertions (Cont.)

```
:- prop sorted/1.

sorted([]).
sorted([_]).
sorted([X,Y|L]):- X =< Y, sorted([Y|L]).

:- regtype list/1.

list([]).
list([_|L]):-
        list(L).
```

Same Example in mixed Logic/Functional Notation

```
:- module(qsort,[qsort/2],[functions,assertions,regtypes]).
:- use_module(library(lists),[append/3]).

:- pred qsort(A,B) : list(A) => sorted(B).
qsort([])    := [].
qsort([X|L]) := ~append(qsort(L1),[X|qsort(L2)]) :- partition(L,X,L1,L2).

:- pred partition(A,B,C,D) : list(A).
partition([],_,[],[]).
partition([E|R],C,[E|Left1],Right):- E < C, !, partition(R,C,Left1,Right).
partition([E|R],C,Left,[E|Right1]):- E >= C,   partition(R,C,Left,Right1).

:- prop sorted/1.
sorted := [] | [_].
sorted := [X,Y|L] :- X=<Y, sorted([Y|L]).

:- regtype list/1. list := [] | [_|~list].
```

Syntactic Sugar

- Lots of syntactic sugar available (always translated to kernel format):
  - Example: 'star' notation for compactness.

    ```
    :- pred p/2 : list(int) * var => list(int) * {int,positive}.
    ```

    is expanded to:

    ```
    :- pred p(A,B) : ( list(A,int), var(B) )
                  => ( list(A,int), int(B), positive(B) ).
    ```

- All the standard Ciao syntactic sugaring can be also be used.
  - Example: using functional notation for defining types:

    ```
    :- regtype color/1.   color := green | blue | red.
    ```

    is expanded to:

    ```
    :- regtype color/1.
    color(green).   color(blue).   color(red).
    ```

    Also, e.g.:

    ```
    :- regtype list/1.   list := [] | [_|~list].
    ```

Mode Definitions ('Property Macros')

- Provide a compact way of expressing properties in assertions.
- Examples of modes (from the basic ISO modes):

```
:- modedef '+'(A) : nonvar(A).
:- modedef '-'(A) : var(A).
:- modedef '@'(A) + not_further_inst(A).
:- modedef '?'(_).
```

- Modes can be used in the argument positions of the $PredPattern$s of assertions.
- The following assertion:

```
:- pred qsort(+,-).
```

  is expanded to:

```
:- pred qsort(X,Y) : nonvar(X), var(Y).
```

Mode Definitions (Cont.)

- Solve the issue of clarifying the meaning of modes.
- Allow defining new modes.
- 'Understood' by documenter (can selectively expand them or not).
- Sets of useful modes are available in the libs:
  - `isomodes` (the ones in the standard),
  - `basicmodes`,
  - etc.

  plus, of course, user-defined modes.

Parametric Mode Definitions

- Parameters can be used in mode definitions.
- Examples of parametric modes (the compound ISO modes):

```
:- modedef +(A,X) :  X(A).
:- modedef @(A,X) :  X(A) => X(A) + not_further_inst(A).
:- modedef -(A,X) :  var(A) => X(A).
:- modedef ?(A,X) :: X(A) => X(A).
```

- The following assertion:

```
:- pred qsort(+list,-list).
```

  is expanded to:

```
:- pred qsort(X,Y) : list(X), var(Y) => list(Y).
```

Documentation

- A documentation field can be added to all predicate assertions.
- In a # preceded field at the end of the assertion.
- A 'classical' entry (typical of traditional Prolog code) such as:

```
%% qsort(+list(int),-list(int)).
%% Argument 2 is a sorted permutation of argument 1.
```

  can be written simply as:

```
:- pred qsort(+list(int),-list(int))
 # "Argument 2 is a sorted permutation of argument 1.".
```

  with the advantage that it is then understood by analyzers, specializers, documenters...

- A documentation-specific assertion also available (`comment/2` / `doc/2`) allows producing full manuals via `lpdoc`.
- See the `lpdoc` documentation for much more...

Program Point Assertions

- Properties of program points between literals in clauses.

  ```
  ..., Literal, check(Cond), Literal, ...
  ```

- Example:

  ```
  p(X) :- q(X,Y), check((Y>=0,list(X,int)), r(Y).
  ```

Assertion "Status" (e.g., *Compiler Output*)

- Assertions can have a prefix: `check`, `true/checked`, `false`, `trust`.
- All previous assertions are "check" (i.e., this is default).
- "True/Checked" assertions: have been proved to hold.
  (e.g., output from the analyzer / assertion checker).
  - Example:

```
:- true success p(X) => ground(X).
```

  - Also, program point output. Example:

```
p(X,Y):-
    true(ground(X)),
    q(X,Z),
    true((ground(X),ground(Z))),
    r(Z,Y),
    true((ground(X),ground(Y),ground(Z))).
```

Assertion "Status" (e.g., *Compiler Output*)

- Trust assertions, to guide compile:

```
:- trust pred is(X,Y) => (num(X),numexpr(Y)).
```

- Assertion *status* summary:
  - `check` (default) – intended semantics, to be checked.
  - `true`, `false` – actual semantics, output from compiler.
  - `trust` – actual semantics, input from user (guiding compiler).
  - `checked` – validation: a `check` that has been proved (same as a `true`).

Guiding Analysis

- "Entry" assertions: describe the calls to a predicate which are *external*:
  - ◇ In general: from outside the module or file.
  - ◇ Also, from meta-predicates (possibly in the module or file).

  Example:

```
:- entry q(X,Y) : (ground(X),var(Y)).
```

- Using "trust" assertions: have to be assumed to hold.
  (e.g., guiding the analyzer / assertion checker).
  - ◇ Example:

```
:- trust success p(X) => ground(X).
```

  - ◇ Predicate, if present, still has to be analyzed.
  - ◇ In some cases, results of analysis may:
    - * improve precision,
    - * or even detect errors in trust declarations.
  - ◇ Very useful also for modular analysis, etc. [ESOP'96]

Compatibilty and *Instantiation*

- If we say an argument "is a list of integers," we must decide if we mean:
  - "The argument is *instantiated* to a list of integers."
    E.g., true for `[]` and `[1,2]`; false for `X`, `[a,2]`, and `[X,2]`.
  - "If any part of the argument is instantiated, this instantiation must be *compatible* with the argument being a list of integers."
    E.g., true for `[]`, `[1,2]`, `X`, and `[X,2]`; false for `[X|1]`, `[a,2]`, and `1`.
- We refer to this as *instantiation properties* vs. *compatibility properties* –both are useful!
- CiaoPP allows us to write properties simply, e.g.:

```
intlist([]).
intlist([X|R]) :- int(X), intlist(R).
```

  and provides the glue code to cover the two cases (inst/1 or compat/1):

  - `compat(X)` $\approx$ `\+ \+ call(X).`
  - `inst(X)` $\approx$ `copy_term(X,Y), call(Y), not_further_instantiated(X,Y).`

Compatibilty and *Instantiation*

- Properties are understood by *default* as *instantiation* properties. E.g.

```
:- pred append(A,B,C) : (intlist(A),intlist(B)) => intlist(C).
```

  means: on calls args 1 and 2 are instantiated to intlists, and arg 3 on success.

- *Compatibilty* can be expressed with the `compat/1` meta-property:

```
:- pred append
   : (compat(intlist(A)),compat(intlist(B)),compat(intlist(C)))
  => (compat(intlist(A)),compat(intlist(B)),compat(intlist(C))).
```

- The following syntactic sugar can be used to express the same as above:

```
:- pred append :: (intlist(A),intlist(B),intlist(C)).
```

  i.e., a `::` field in assertions can be used for compat props.

Some Final Comments

- If in doubt about the meaning of an assertion: run the documenter!
  (just push the documentation generation button while in the source file)
- Processors (analyzers, specializers, etc.) only need to understand the basic assertions – all syntactic sugar (including modes, star notation, functional notation, etc.) is removed.

The Ciao Preprocessor (CiaoPP)

CiaoPP: The *Ciao* System Preprocessor

- *CiaoPP* [25, 28] 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 [42] & syntactic extensions).
  - Output: *error/warning messages* + *transformed logic program*, with
    - Results of analysis (as assertions).
    - Results of static checking of assertions [26, 41] / verification.
    - Certificates for Abstraction Carrying Code.
    - Assertion run-time checking code.
    - Optimizations (specialization, slicing, parallelization, low-level optim., etc.).
- Generic tool – can be applied to other systems.
- Underlying technology:
  - Modular polyvariant abstract interpretation [6, 31].
  - Modular abstract multiple specialization [44].
- See specific tutorial on the CiaoPP system.

Other Issues

Other Issues

- FD constraint solver.
- Full implementation of distribution [16].
- Fully transparent access to databases (bypassing schizoid DB characteristics).
- Compilation to C [36], abstract machine specialization.
- Making expansion rule order transparent.
- KUL CHR [20] supported.
- Improved delay (`when`, `freeze`, ...) primitives.
- Efficient constructive negation [39].

+ Many issues related to program analysis and transformation (CiaoPP).

- ...

External Collaborations and Funding

- Ciao/CiaoPP has been developed so far in collaboration with: G. Gupta (UT Dallas), E. Pontelli (NM State University), P. Stuckey and M. García de la Banda (Melbourne U.), K. Marriott (Monash U.), M. Bruynooghe, A. Mulkers, G. Janssens, and V. Dumortier (K.U. Leuven), S. Debray (U. of Arizona), J. Maluzynski and W. Drabent (Linkoping U.), P. Pietrzak (UPM), P. Deransart (INRIA), J. Gallagher (Roskilde University), C. Holzbauer (Austrian Research Institute for AI), M. Codish (Beer-Sheva), S. Genaim (Beer-Sheva/UPM), SICS, ...
- Ciao/CiaoPP has been supported so far in part by:
  - EU/ESPRIT projects MOBIUS, ASAP, AMOS, ACCLAIM, PARFORCE, PRINCE, DISCIPL, and RadioWeb.
  - CICYT/MCYT grants IPL-D, ELLA, EDIPIA, CUBICO, and MERIT.
  - ESPRIT Networks of Excellence Compulog II/IV and CoLogNet
  - US/EU Fulbright collaboration grant ECCOSIC, ADELA Spanish/Italian Integrated Action.
  - Motorola Inc.

Downloading the system(s)

- Downloading `ciao`, `ciaopp`, `lpdoc`, and other CLIP software:
  - Standard distributions:
    `http://www.clip.dia.fi.upm.es/Software`
  - Some betas (in testing or completing documentation – ask `webmaster` for info) in:
    `http://www.clip.dia.fi.upm.es/Software/Beta`
  - 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`
    Mail list stored in
    `http://www.clip.dia.fi.upm.es/Mail/ciao-users/`

Some Bibliography on Ciao, CiaoPP, and LPdoc

[1] E. Albert, P. Arenas, G. Puebla, and M. Hermenegildo. Reduced Certificates for Abstraction-Carrying Code. In *22nd International Conference on Logic Programming (ICLP 2006)*, number 4079 in LNCS, pages 163–178. Springer-Verlag, August 2006.

[2] E. Albert, G. Puebla, and M. V. Hermenegildo. Abstraction-Carrying Code. In *Proc. of LPAR'04*, volume 3452 of *LNAI*. Springer, 2005.

[3] F. Bueno, D. Cabeza, M. Carro, M. V. Hermenegildo, P. Lopez-Garcia, and G. Puebla. The Ciao Prolog System. Reference Manual (v1.8). The Ciao System Documentation Series–TR CLIP4/2002.1, School of Computer Science, Technical University of Madrid (UPM), May 2002. System and on-line version of the manual available at `http://ciao-lang.org`.

[4] F. Bueno, D. Cabeza, M. V. 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.

[5] F. Bueno, M. García de la Banda, M. V. Hermenegildo, K. Marriott, G. Puebla, and P. Stuckey. A Model for Inter-module Analysis and Optimizing Compilation. In *Logic-based Program Synthesis and Transformation*, number 2042 in LNCS, pages 86–102. Springer-Verlag, March 2001.

[6] F. Bueno, M. García de la Banda, and M. V. 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.

[7] I. Caballero, D. Cabeza, S. Genaim, J.M. Gomez, and M. V. Hermenegildo. persdb_sql: SQL Persistent Database Interface. Technical Report CLIP10/98.0, December 1998.

[8] D. Cabeza and M. Hermenegildo. Distributed WWW Programming using (Ciao) Prolog and the PiLLoW Library. *Theory and Practice of Logic Programming*, 1(3):251–282, May 2001.

[9] D. Cabeza, M. Hermenegildo, and S. Varma. The PiLLoW/Ciao Library for INTERNET/WWW Programming using Computational Logic Systems, May 1999. See http://www.cliplab.org/Software/pillow/pillow.html.

[10] D. Cabeza and M. V. Hermenegildo. Implementing Distributed Concurrent Constraint Execution in the CIAO System. In *Proc. of the AGP'96 Joint conference on Declarative Programming*, pages 67–78, San Sebastian, Spain, July 1996. U. of the Basque Country. Available from `http://www.cliplab.org/`.

[11] D. Cabeza and M. V. Hermenegildo. WWW Programming using Computational Logic Systems (and the PiLLoW/Ciao Library). In *Proceedings of the Workshop on Logic Programming and the WWW at WWW6*, San Francisco, CA, April 1997.

[12] D. Cabeza and M. V. Hermenegildo. A New Module System for Prolog. In *International Conference on Computational Logic, CL2000*, number 1861 in LNAI, pages 131–148. Springer-Verlag, July 2000.

[13] D. Cabeza and M. V. Hermenegildo. The Ciao Modular, Standalone Compiler and Its Generic Program Processing Library. In *Special Issue on Parallelism and Implementation of (C)LP Systems*, volume 30(3) of *Electronic Notes in Theoretical Computer Science*. Elsevier - North Holland, March 2000.

[14] D. Cabeza, M. V. Hermenegildo, and J. Lipton. Hiord: A Type-Free Higher-Order Logic Programming Language with Predicate Abstraction. In *Ninth Asian Computing Science Conference (ASIAN'04)*, number 3321 in LNCS, pages 93–108. Springer-Verlag, December 2004.

[15] M. Carro and M. Hermenegildo. Concurrency in Prolog Using Threads and a Shared Database. In *1999 International Conference on Logic Programming*, pages 320–334. MIT Press, Cambridge, MA, USA, November 1999.

[16] M. Carro and M. Hermenegildo. A simple approach to distributed objects in prolog. In *Colloquium on Implementation of Constraint and LOgic Programming Systems (ICLP associated workshop)*, Copenhagen, July 2002.

[17] A. Casas, D. Cabeza, and M. V. Hermenegildo. A Syntactic Approach to Combining Functional Notation, Lazy Evaluation and Higher-Order in LP Systems. In *The 8th International Symposium on Functional and Logic Programming (FLOPS'06)*, pages 142–162, Fuji Susono (Japan), April 2006.

[18] S. K. Debray, P. Lopez-Garcia, M. V. Hermenegildo, and N.-W. Lin. Lower Bound Cost Estimation for Logic Programs. In *1997 International Logic Programming Symposium*, pages 291–305. MIT Press, Cambridge, MA, October 1997.

[19] S.K. Debray, P. Lopez-Garcia, and M. V. Hermenegildo. Non-Failure Analysis for Logic Programs. In *1997 International Conference on Logic Programming*, pages 48–62, Cambridge, MA, June 1997. MIT Press, Cambridge, MA.

[20] Thom Frühwirth. Theory and Practice of Constraint Handling Rules. *Journal of Logic Programming*, 37(1-3), October 1998.

[21] J.M. Gomez, D. Cabeza, and M. V. Hermenegildo. persdb: Persistent Database Interface. Technical Report CLIP9/98.0, December 1998.

[22] M. Hermenegildo, D. Cabeza, and M. Carro. Using Attributed Variables in the Implementation of Concurrent and Parallel Logic Programming Systems. In *Proc. of the Twelfth International Conference on Logic Programming*, pages 631–645. MIT Press, June 1995.

[23] M. V. Hermenegildo. A Documentation Generator for (C)LP Systems. In *International Conference on Computational Logic, CL2000*, number 1861 in LNAI, pages 1345–1361. Springer-Verlag, July 2000.

[24] M. V. Hermenegildo, F. Bueno, D. Cabeza, M. Carro, M. García de la Banda, P. Lopez-Garcia, and G. Puebla. The CIAO Multi-Dialect Compiler and System: An Experimentation Workbench for Future (C)LP Systems. In *Parallelism and Implementation of Logic and Constraint Logic Programming*, pages 65–85. Nova Science, Commack, NY, USA, April 1999.

[25] M. V. Hermenegildo, F. Bueno, G. Puebla, and P. Lopez-Garcia. Program Analysis, Debugging and Optimization Using the Ciao System Preprocessor. In *1999 Int'l. Conference on Logic Programming*, pages 52–66, Cambridge, MA, November 1999. MIT Press.

[26] M. V. 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.

[27] M. V. Hermenegildo, G. Puebla, F. Bueno, and P. Lopez-Garcia. Abstract Verification and Debugging of Constraint Logic Programs. In *Recent Advances in Constraints*, number 2627 in LNCS, pages 1–14. Springer-Verlag, January 2003.

[28] M. V. Hermenegildo, G. Puebla, F. Bueno, and P. Lopez-Garcia. Program Development Using Abstract Interpretation (and The Ciao System Preprocessor). In *10th International Static Analysis Symposium (SAS'03)*, number 2694 in LNCS, pages 127–152. Springer-Verlag, June 2003.

[29] M. V. Hermenegildo, G. Puebla, F. Bueno, and P. Lopez-Garcia. Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). *Science of Computer Programming*, 58(1–2):115–140, October 2005.

[30] M. V. Hermenegildo, G. Puebla, K. Marriott, and P. Stuckey. Incremental Analysis of Logic Programs. In *International Conference on Logic Programming*, pages 797–811. MIT Press, June 1995.

[31] M. V. Hermenegildo, G. Puebla, K. Marriott, and P. Stuckey. Incremental Analysis of Constraint Logic Programs. *ACM Transactions on Programming Languages and Systems*, 22(2):187–223, March 2000.

[32] C. Holzbaur. Metastructures vs. Attributed Variables in the Context of Extensible Unification. In *Int'l. Symposium on Programming Language Implementation and Logic Programming*, number 631 in LNCS, pages 260–268. Springer Verlag, Aug 1992.

[33] C. Holzbaur. *SICStus 2.1/DMCAI Clp 2.1.1 User's Manual*. University of Vienna, 1994.

[34] P. Lopez-Garcia and M. V. Hermenegildo. Efficient Term Size Computation for Granularity Control. In *International Conference on Logic Programming*, pages 647–661, Cambridge, MA, June 1995. MIT Press, Cambridge, MA.

[35] P. Lopez-Garcia, M. V. Hermenegildo, and S. K. Debray. A Methodology for Granularity Based Control of Parallelism in Logic Programs. *Journal of Symbolic Computation, Special Issue on Parallel Symbolic Computation*, 21(4–6):715–734, 1996.

[36] J. Morales and M. Carro. Improving the Compilation of Prolog to C Using Type Information: Preliminary Results. In M. Carro, C. Vaucheret, and K.-K. Lau, editors, *Proceedings of the CBD 2002 / ITCLS 2002 CoLogNet Joint Workshop*, pages 167–180, School of Computer Science, Technical University of Madrid, September 2002. Facultad de Informatica.

[37] J. Morales, M. Carro, G. Puebla, and M. Hermenegildo. A Generator of Efficient Abstract Machine Implementations and its Application to Emulator Minimization. In Maurizio Gabbrielli and Gopal Gupta, editors, *International Conference on Logic Programming*, number 3668 in LNCS, pages 21–36. Springer Verlag, October 2005.

[38] J. Navas, F. Bueno, and M. V. Hermenegildo. Efficient Top-Down Set-Sharing Analysis Using Cliques. In *8th International Symposium on Practical Aspects of Declarative Languages (PADL'06)*, number 2819 in LNCS, pages 183–198. Springer-Verlag, January 2006.

[39] S. Mu noz, J.J. Moreno-Navarro, and M. V. Hermenegildo. Efficient Negation Using Abstract Interpretation. In *Proc. of the Eigth International Conference on Logic Programming and Automated Reasoning*, LNAI. Springer-Verlag, December 2001.

[40] A. Pineda and M. Hermenegildo. O'Ciao: An Object Oriented Programming Model for (Ciao) Prolog. Technical Report CLIP 5/99.0, Facultad de Informática, UPM, July 1999.

[41] G. Puebla, F. Bueno, and M. V. Hermenegildo. A Generic Preprocessor for Program Validation and Debugging. In P. Deransart, M. V. Hermenegildo, and J. Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming*, number 1870 in LNCS, pages 63–107. Springer-Verlag, September 2000.

[42] G. Puebla, F. Bueno, and M. V. Hermenegildo. An Assertion Language for Constraint Logic Programs. In P. Deransart, M. V. Hermenegildo, and J. Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming*, number 1870 in LNCS, pages 23–61. Springer-Verlag, September 2000.

[43] G. Puebla, F. Bueno, and M. V. Hermenegildo. Combined Static and Dynamic Assertion-Based Debugging of Constraint Logic Programs. In *Logic-based Program Synthesis and Transformation (LOPSTR'99)*, number 1817 in LNCS, pages 273–292. Springer-Verlag, March 2000.

[44] G. Puebla and M. V. Hermenegildo. Abstract Multiple Specialization and its Application to Program Parallelization. *J. of Logic Programming. Special Issue on Synthesis, Transformation and Analysis of Logic Programs*, 41(2&3):279–316, November 1999.

[45] G. Puebla and M. V. Hermenegildo. Abstract Specialization and its Applications. In *ACM Partial Evaluation and Semantics based Program Manipulation (PEPM'03)*, pages 29–43. ACM Press, June 2003. Invited talk.