Computational Logic:
(Constraint) Logic Programming

Theory, practice, and implementation

The *Ciao* Logic Programming Environment – A Tutorial –

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

Introduction: The Ciao Program Development System

- Ciao is a next-generation (C)LP programming environment – features:
  - ◇ Free software (GNU license).
  - ◇ Designed to be *extensible* and *analyzable*.
  - ◇ Pure kernel (*no "built-ins"*); subsumes ISO-Prolog (transparently) via *library*.
  - ◇ 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, ...
  - ◇ Support for several paradigms:
    - * functions, higher-order, objects, constraint domains, ...
    - * concurrency, parallelism, distributed execution, ...
  - ◇ Advanced Emacs environment (with e.g., automatic access to documentation).

Introduction: The Ciao Program Development System (Contd.)

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

`ciaosh`:	Standard top-level shell.
`ciaoc`:	Standalone compiler.
`ciao-shell`:	Script interpreter.
`lpdoc`:	Documentation generator (info, ps, pdf, html, ...).
`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 Module System

- Ciao implements a module system [6] which meets a number of objectives:
  - ◇ High extensibility in syntax and functionality.
  - ◇ Amenability to modular (separate) processing of program components.
  - ◇ Amenability to (modular) global analysis.
  - ◇ Greatly enhanced error detection (e.g., undefined predicates).
  - ◇ Support for meta-programming and higher-order.
  - ◇ Compatibility with official and de-facto standards.
  - ◇ Backward compatible with files which are not modules.

The Ciao Module System (Contd.)

- Some more specific characteristics [6]:
  - ◇ 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 [7]:
  - ◇ 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.

Linking Regimes of Executables

- 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.

The Ciao-Emacs Development Environment

- Customized program editor:
  In one (simple and powerful) word: Emacs!!
- A Ciao mode (`ciao.el`) including:
  - ◇ Incremental syntax highlighting of source code.
  - ◇ Direct access to on-line documentation (help on what the cursor is on).
  - ◇ Menu-guided support for compilation, top-level, preprocessing, ...
  - ◇ Also to *generation* of documentation.
  - ◇ Location of errors from compiler (and preprocessor) on source code.
  - ◇ Support for source code debugging.
  - ◇ Plus many other features!

Ciao Mode Menu

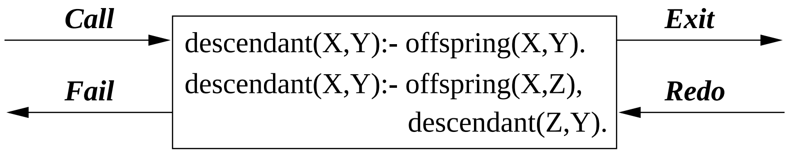
HELP	
Help (for symbol under cursor)	(C-c TAB)
Complete symbol under cursor	(C-c /)
Ciao system manual	
List all key bindings	
TOP-LEVEL/COMPILER	
Start ciao top level	(C-c t)
(Re)Load buffer into top level	(C-c l)
(Re)Load all modules as necessary	(C-c L)
Locate (next) preproc/compiler error msg	(C-c ‘)
Remove preproc/compiler error mark	(C-c e)
Query and main file	
Check buffer syntax (incl. assertions)	(C-c E)
Make executable from buffer	(C-c x)
Make object file (.po) from buffer	(C-c o)
Make active module from buffer	(C-c a)
Insert script header	(C-c I S)

Ciao Mode Menu (Contd.)

TOP-LEVEL/DEBUGGER	
(Un)Debug buffer source	(C-c d)
Select debug mode	(C-c m)
Select multiple buffers for debug	(C-c M-m)
Breakpoints	
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)
SET MODE DEFAULTS)	
TRADITIONAL PROLOG COMMANDS (also for SICStus)	
Ciao/Prolog mode version	(C-c v)

Debugging in Ciao

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



Call

Fail

descendant(X,Y):- offspring(X,Y).
descendant(X,Y):- offspring(X,Z),
descendant(Z,Y).

Exit

Redo

```
+  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).

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).
```

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).
- 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 [22]), . . .

Package Example: Functional Notation (functions)

- Functional notation: defined via (local) ops/expansions.
  Provides simple syntactic translation to predicate definitions/calls:
  - ◇ Defining function `F/N` implies defining predicate `F/(N+1)`.
  - ◇ Any predicate `P/(N+1)` can be used as a function `P/N`.
  - ◇ The last argument of the predicate holds the result of the function.
- Using functions does not incur in run-time slowdowns.
- Function applications are recognized by:
  - ◇ Marking them with the prefix operator ~
  - ◇ A `:- function(F/N)` declaration makes all occurrences of a `F/N` functor to be regarded as a function call.
  - ◇ Arithmetic operators are translated to `is/2` calls (optional).
- Example: defining `fact/1` factorial function (and `fact/2` predicate!)

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

Functions Package: Examples – Array Access

- Assume multi-dimensional arrays such as:

```
:- use_package(functions).

array([N],A) :-
        functor(A,a,N).
array([N|Ms],A) :-
        functor(A,a,N),
        rows(N,Ms,A).

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

Functions Package: Examples – Array Access

- We can now define the array access function with some syntactic sugar:

```
:- op(45, xfx, [@]).
:- function '@'/2 .

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

- And use it: `?- array([2,2],M), M@[2,1] = 3, display(M).`
- E.g., in a vector addition:

```
vecplus(V1,V2,V3) :-
        array([N],V1), array([N],V2), array([N],V3), vecplus_(N,V1,V2,V3).

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

Functions Package: Examples – Sugar for Append

- Some syntactic sugar for append:

```
:- function append/2.
test1 :- set_prolog_flag(write_strings,on),
         X = " ",
         write(append("Hello",append(X,"world!"))).
```

- Even more:

```
:- function append/2.
:- op(200,xfy,[::]).
:- function :: /2.

A :: B := append(A,B).

test2 :- set_prolog_flag(write_strings,on),
         X = " ",
         write("Hello" :: X :: "world!").
```

Higher-Order Programming

- HO in Ciao [5] currently implemented by the family of `call/N` builtins.
- First argument of `call/N` may be instantiated to:
  - ◇ A higher-order term, supporting currying: `member([1,2])`
  - ◇ A "predicate abstraction": `(''(X,Y) :- Y is X+10)`
- Currying adapted to familiar semantic conventions in logic programs:
  - ◇ For unary higher-order predicates, the argument is added first:

    `>(0)` checks if a given number is *greater than* `0`

    `member(L)` is true if the argument is member of the list `L`
  - ◇ For binary higher-order predicates, the first argument is added first, the second one is added last:

    `append(".")` appends to the first argument the string `"."`, returning the result in the second one

    `/(2)` divides by 2 the first argument, returning the result in the second one (given the definition `/(X,Y,Z) :- Z is X/Y`)

Higher-Order Programming (Contd.): Currying Rules in Action

```
:- set_prolog_flag(multi_arity_warnings,off).

p(X,Y,Z,W,K,L) :- display(p(X,Y,Z,W,K,L)),nl.


?- use_package(hiord).

?- call(p,1,2,3,4,5,6).    -->    p(1,2,3,4,5,6)
?- call(p(1),2,3,4,5,6).  -->    p(2,1,3,4,5,6)
?- call(p(1,2),3,4,5,6).  -->    p(3,1,2,4,5,6)
?- call(p(1,2,3),4,5,6).  -->    p(4,1,2,3,5,6)
?- call(p(1,2,3,4),5,6).  -->    p(5,1,2,3,4,6)
?- call(p(1,2,3,4,5),6).  -->    p(6,1,2,3,4,5)
?- call(p(1,2,3,4,5,6)).  -->    p(1,2,3,4,5,6)
```

Higher-Order Programming (Contd.)

- Special syntax supported:
  `P(X,...)` is read as `call(P,X,...)`,
  `_(X,...)` is read as `''(X,...)`
- `meta_predicate/1` declarations extended to reflect higher-order predicates.
- Example: parametric list regular type:

```
:- prop list(L,T) + regtype # "@var{L} is a list of @var{T}s.".
:- meta_predicate list(?, pred(1)).
list([],_).
list([X|Xs], T) :-
        T(X),
        list(Xs, T).
```

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

Combining Higher-Order with Functional Notation

- They can be combined, resulting in essentially functional-style code (but sometimes with additional modes of use!).
- Some common examples (in `library(hiordlib)`):

```
:- 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)).
```

- Example use of `map/3`: translating a list of indices to a list of terms.

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

Records package (named arguments)

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

```
:- include(library(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),_)).
```

Using Other Computation Rules

- Libraries which replace the depth-first, left to right computation rule of Prolog.
- Compile-time transforming programs (Compiling Control technique).
- Useful in search problems when a complete proof procedure is needed. (e.g., for teaching logic)
- Computation rules currently implemented:
  - ◇ Breadth-first (`bf` package)
  - ◇ "And-fair" breadth-first (`bf/af` package)
  - ◇ Iterative-deepening (`id` package)
  - ◇ Depth-First search with limited depth (`id` package)
  - ◇ Mycin.
  - ◇ Andorra (in Beta).

Breadth-First

- Use package `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 `bf/af` version ensures “And-fairness” by goal shuffling.

Breadth-First Example I

```
:- module(chain, [test/1], [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(sublist, [test/1], ['bf/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 behaviour.
- 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).
```

Constraints

- Currently two packages: `clpq` and `clpr`.
- Based on Holzbaur's implementation [19, 18] using attributed variables.
- The effect 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

- 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).
```

Persistent Predicates

- Persistent Predicate [10, 2]: dynamic predicate residing in non-volatile media.
- Its state survives across successive executions of the application.
- Currently supported storage media:
  - ◇ Files: `persdb` package.
  - ◇ SQL database: `persdb_sql` package.
- 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 analog to `assert/1` and `retract/1`.
- Transactional behaviour.

Persistent Predicates Example

- 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.
```

Object-Oriented Features: O'Ciao

- Basic design philosophy [20]:
  - ◇ 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	Correspondance in (Ciao) Prolog
State	attributes	dynamic predicates
Encapsulation	classes	modules
Polymorphism	overloading	clause selection
Instantiation	objects	–
Inheritance	classes	(reexport)

- Missing is instantiation → 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],[objects]).
:- 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.
- (Implemented basically on top of Standard Prolog.)

- Similar capabilities to other designs (e.g., SICStus objects, Logtalk, ...). But those are typically unrelated to the module structure.

O'Ciao: Defining Classes

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

  `:- class(stack).` ≡ `:- 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).

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).
```

Basic Concurrency

- (Low-level) Concurrency in Ciao Prolog is currently provided [9] 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 (note the `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 ⟶ 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 [14, 3] 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 <&, ...
```

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

```
A & B :- A &> Ha, B, Ha <& .
```

Distributed Execution

- It is very easy to write a server which listens on a port for goals and executes them [3] (recall the TCP/IP-based server):

```
:- concurrent conn/1.              handle_query:-
                                        retract_fact(conn(Stream)),
wait_for_queries(Socket):-              read(Stream, Goal),
     repeat,                            call(Goal),
     socket_accept(Socket, Stream),     write(Stream, Goal),
     assertz_fact(conn(Stream)),        close(Stream),
     fail.                              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 [3] (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.

Active Objects, Code, and Computation Mobility

- Code mobility is easy: code just a set of Prolog 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 ⟶ set of Prolog 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

- The PiLLoW library simplifies the process of writing Internet and WWW applications [8, 4].
- 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.

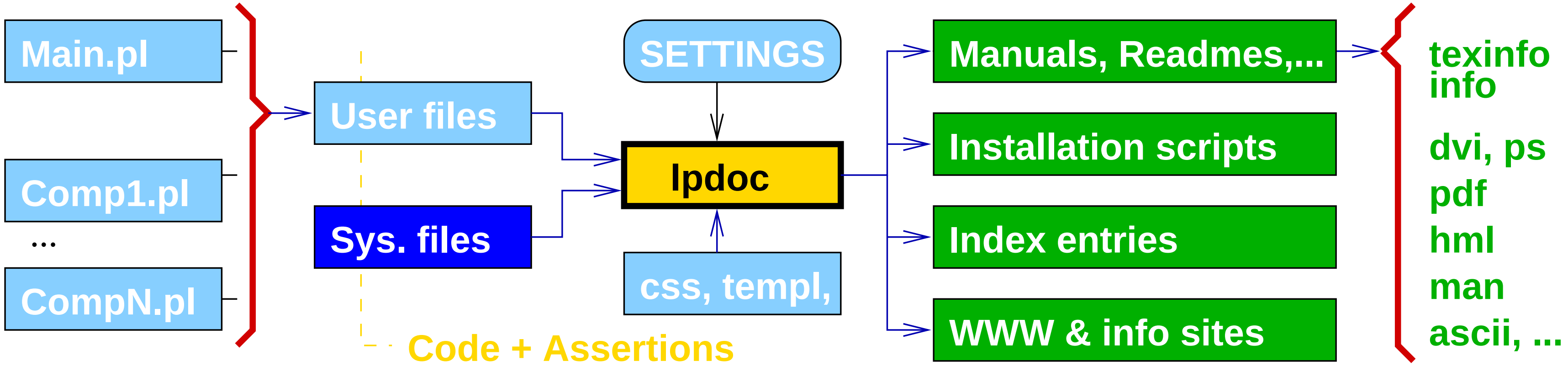
Form Producer/Handler Example

```
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').
(...)
```

LPdoc: the Ciao Automatic Documentation System

- Writing and, specially, maintaining program documentation is hard
  → 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 [22].
  - ◇ 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.
- `LPdoc` [11] understands assertions natively and uses them to generate the documentation.

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(CommentType,CommentData).
```

- Examples:

```
:- comment(title,"Complex numbers library").
:- comment(summary,"Provides an ADT for complex numbers.").
:- comment(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 → 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` → assumed *library*, *interface* (API) documented. else → 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).

CiaoPP: The *Ciao* System Preprocessor

- *CiaoPP* [13, 16] is a standalone preprocessor to the standard clause-level compiler.
- Performs source-to-source transformations:
  - ◇ Input: logic program (optionally w/assertions [22] & syntactic extensions).
  - ◇ Output: *error/warning messages* + *transformed logic program*, with
    - * Results of analysis (as assertions).
    - * Results of static checking of assertions [15, 21].
    - * Assertion run-time checking code.
    - * Optimizations (specialization, parallelization, etc.)
- By design, a generic tool – can be applied to other systems (e.g., CHIP → CHIPRE).
- Underlying technology:
  - ◇ Modular polyvariant abstract interpretation [1, 17].
  - ◇ Modular abstract multiple specialization [23].

External Collaborations and Funding

- Ciao/CiaoPP has been developed so far in collaboration with:
  G. Gupta and 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, P. Pietrzak and W. Drabent (Linkoping U.), P. Deransart (INRIA), J. Gallagher (Bristol University), C. Holzbauer (Austrian Research Institute for AI), M. Codish, S. Genaim (Beer-Sheva), SICS.
- Ciao/CiaoPP has been supported so far in part by:
  - ◇ EU/ESPRIT projects PARFORCE, PRINCE, ACCLAIM, DISCIPL, and RadioWeb.
  - ◇ CICYT/MCYT grants IPL-D, ELLA, EDIPIA, and CUBICO.
  - ◇ US/EU Fulbright collaboration grant ECCOSIC.

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/`

Recent Bibliography on Ciao, CiaoPP, and LPdoc

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

[2] I. Caballero, D. Cabeza, S. Genaim, J.M. Gomez, and M. Hermenegildo. persdb˙sql: SQL Persistent Database Interface. Technical Report CLIP10/98.0, December 1998.

[3] D. Cabeza and M. 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.clip.dia.fi.upm.es/`.

[4] D. Cabeza and M. 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.

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

[6] D. Cabeza and M. 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.

[7] D. Cabeza and M. 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 of *Electronic Notes in Theoretical Computer Science*. Elsevier - North Holland, March 2000.

[8] D. Cabeza, M. Hermenegildo, and S. Varma. The PiLLoW/Ciao Library for INTERNET/WWW Programming using Computational Logic Systems, May 1999. See http://www.clip.dia.fi .upm.es/Software/pillow/pillow.html.

[9] 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.

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

[11] M. 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.

[12] M. Hermenegildo, F. Bueno, D. Cabeza, M. Carro, M. García de la Banda, P. López-García, 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.

[13] M. Hermenegildo, F. Bueno, G. Puebla, and P. López-García. Program Analysis, Debugging and Optimization Using the Ciao System Preprocessor. In *1999 International Conference on Logic Programming*, pages 52–66, Cambridge, MA, November 1999. MIT Press.

[14] 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.

[15] M. Hermenegildo, G. Puebla, and F. Bueno. Using Global Analysis, Partial Specifications, and an Extensible Assertion Language for Program Validation and Debugging. In K. R. Apt, V. Marek, M. Truszczynski, and D. S. Warren, editors, *The Logic Programming Paradigm: a 25–Year Perspective*, pages 161–192. Springer-Verlag, July 1999.

[16] M. Hermenegildo, G. Puebla, F. Bueno, and P. López-García. 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.

[17] M. 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.

[18] C. Holzbaur. Metastructures vs. Attributed Variables in the Context of Extensible Unification. In *1992 International Symposium on Programming Language Implementation and Logic Programming*, pages 260–268. LNCS631, Springer Verlag, August 1992.

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

[20] 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.

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

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

[23] G. Puebla and M. 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.