# Internet and WWW Programming using Computational Logic Systems (PART II)

Manuel Hermenegildo
(joint work with Daniel Cabeza)
{herme,dcabeza}@dia.fi.upm.es

http://www.clip.dia.fi.upm.es/

*School of Computer Science*
*Technical University of Madrid (UPM)*

*Workshop on Logic Programming and the Internet*
December 19–20, 1996
Imperial College, London, U.K.

Work supported in part by ESPRIT projects "ACCLAIM" and "PARFORCE" and by CICYT/MEC grant "ELLA." Sponsored by Compulog-Net III.

# Outline

- (PART I: *WWW programming*)
- PART II: *Distributed (and agent) programming*
  - Blackboards.
  - Modeling and accessing information servers (active modules).
  - A simple distributed LP/CLP language using “worker teams”.
  - Implementing distributed variable-based communication using attributed variables.
- Different distributed execution scenarios:
  - Distributed networks of concurrent, communicating agents
  - Request/provide remote services in a distributed network (including database servers, WWW servers, etc.)
  - Coarse-grained Parallelism (granularity control required)
- Most functionality can be obtained using current LP/CLP systems!
- Again, concurrency in the underlying engine is very useful.

## Implementing and Using Blackboards

- Blackboards + linda stile interaction: basic but very useful means of communication and synchronization.
- Present in many systems: SICStus, BinProlog/$\mu^2$-Prolog, &-Prolog/CIAO, ...
- Basic features:
  - `out`/1: write tuple
  - `rd`/1: read tuple
  - `in`/1: remove tuple
  - `rd_noblock`/1 and `in_noblock`/1
  - `in`/2 and `rd`/2 (on disjunctions)
- Possibly, have several (possibly hierarchical) blackboards – then, extra argument to primitives specifies which blackboard. Also, specification of location of blackboard, creation of clients, etc.
- Implementation approaches:
  - Blackboard is a Prolog process. Tuples maintained via assert/retract. Communication, e.g., via sockets (allows Internet-wide use of the blackboard).
  - Support blackboard internally in system (possibly, in conjunction with asserted database).
  - Mixed approach: local vs. remote blackboards.
  - Clients can be created using standard process creation primitives (e.g., "sh" and "rsh"), or dedicated port servers.

# Producer–Consumer: Linda Version

*(using CIAO BB primitives)*

```
?- create_bb(B,local), N=10,
   lproducer(N,B) @ alba &, lconsumer(B).

lproducer(N,B) :-
        lproducer(N,1,B).

% second argument is message order
lproducer(0,C,B) :- !,
   linda:out(message(end(C)),B).
lproducer(N,C,B) :-
   N>0,
   linda:out(message(C,N),B),
   N1 is N-1,
   C1 is C+1,
   lproducer(N1,C1,B).

lconsumer(B) :-
   lconsumer(1,B).

lconsumer(C,B) :-
   linda:rd([message(end(C)),
             message(_,C)], T, B),
   lconsumer_data(T,B).

lconsumer_data(message(end(_)),B).
lconsumer_data(message(N,C),B) :-
   C1 is C+1,
   lconsumer(C1,B).
```

# 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 using a special form of save:
  `:- save_active_module(Filename, Addr, Report_Addr).`
  (`Report_Addr` is code to report `Addr`)
- Procedures can be imported from remote "active modules" via a simple declaration: E.g.
  `:- use_active_module(Name, [P1/N1, P2/N2,...]).`
- A hook predicate to locate active modules (configurable):
  `module_address(Name, Addr)`
- Calls to such imported procedures are executed remotely in a transparent way.
- Example application: client-server. Client imports module which exports the functionality provided by server. Access is transparent from then on.
- Typical examples of active objects: databases, name servers, agendas, WWW form handlers, WWW cgi-bin servers, demons (ftp, telnet, ...), etc.
- Implementation: built as an abstraction on top of ports/sockets (can also be easily built on top of a blackboard).

# Using Active Modules: An Example

Server:

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

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

:- save_active_module(dbserver,
```

Client:

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




replenish(P) :-
       stock(P, S),
       S < 5, ...
```

# Using Active Modules: An Example

```
                                        alba:888, true).
```

```
module_address(database, alba:888).
```

# Using Active Modules: An Example

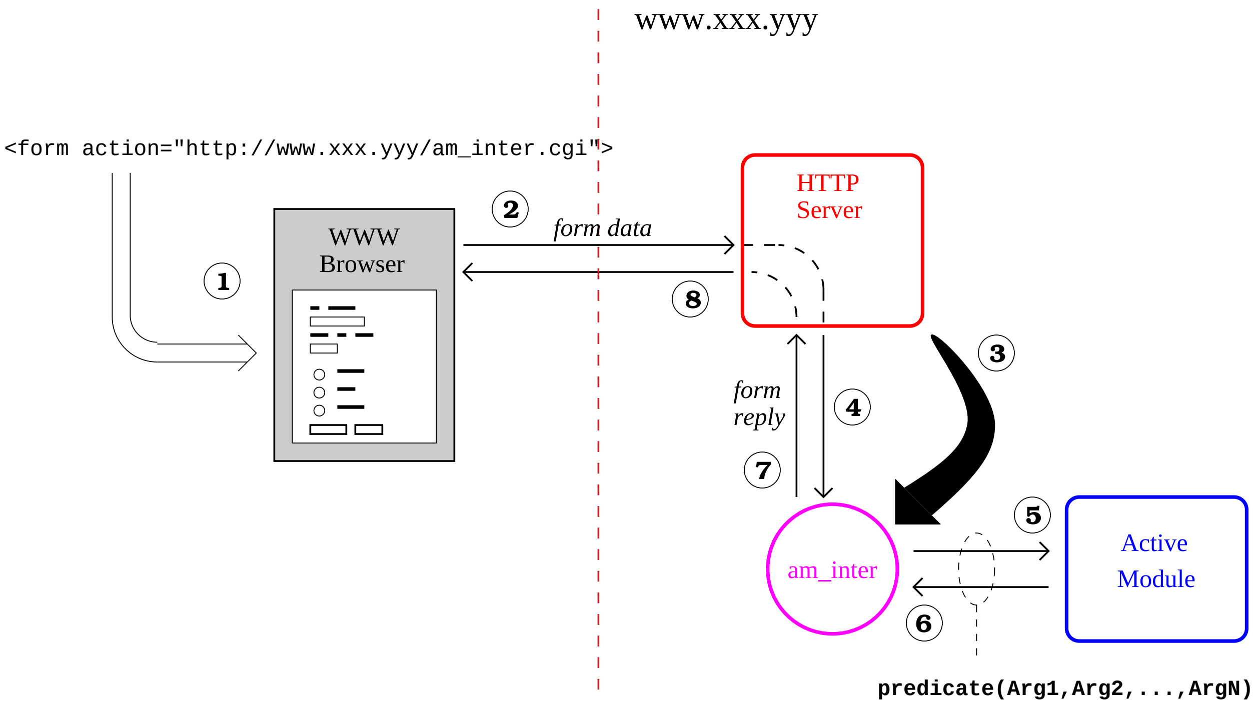
```
:- use_active_module(name_server, [add_address/2]).
```

```
                                    Addr,
                         add_address(database,Addr)).
```

```
:- use_module(locate_module_addresses).
```

# Application: Active Modules as Form Servers

- The cgi-bin interface dictates that a handler of a form starts and terminates for each interaction.
- Thus, form handlers cannot (in principle) have state.
- Active modules can be used to implement persistent form handlers.
- Also, faster than starting the application every time.



# Active Modules as Form Servers: Example

- Active Module:

```
:- module(phone_db,[response/2,add_phone/2]).

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

add_phone(Name, Phone) :-
   assert(phone(Name, Phone)).

%% Database
:- dynamic phone/2.
phone('CLIP',   '336-7448').
phone('Paco',   '554-5225').
phone('Daniel', '460-0569').

export_address(Address) :-
    tell('phone_db.addr'), write(Address), told.

:- save_active_module(phone_db, Address,
                      export_address(Address)).
```

# Active Modules as Form Servers: Example (Cont.)

- CGI script using that active module:

```
#!/usr/local/bin/lpshell
:- use_active_module(phone_db,[response/2]).
:- use_module('/usr/local/src/pillow/pillow.pl').

main(_) :-
        get_form_input(Input),
        get_form_value(Input,person_name,Name),
        response(Name,Response),
        output_html([
          cgi_reply,
          start,
          title('Simple CLIP telephone database'),
          center( [
            image('http://localhost/images/clip.gif'),
            heading(2,'Simple CLIP telephone database'),
            --,
            Response,
            start_form,
            'Click here, enter name of clip member,
            and press Return:',
            \\,
            input(text,[name=person_name,size=20]),
            --,
            end_form,
            image('http://localhost/images/pillow_d.gif') ]),
          end ]).

module_address(phone_db,Address) :-
   see('phone_db.addr'), read(Address), seen.
```

# Distributed Teams of Workers (CIAO)

- Team: set of workers that share the same code and cooperate to run it
- Concurrency and/or parallelism between workers
- Worker management:
  - `add_worker` Add (possibly remote) worker to the team
  - `delete_worker` Delete (possibly remote) worker from the team
- The workers are kept coherent from the point of view of code management, global state, etc.

# Some Concurrency & Parallelism Operators (CIAO)

- Objective: express (independent) And-parallelism, concurrency (dependent And-Parallelism), and fairness
- Independent And-Parallelism:
  - `A &> H` – sends out goal `A`, returns in `H` a handler
  - `H <&` – gets results of goal pointed by `H`, or executes it
  - `A & B` – performs a parallel "fork" of `A` and `B`, waits for the execution of both to finish
  - Last one can be implemented using the other two:

    ```
    A & B :- B &> H, call(A), H <& .
    ```
  - The bindings made in the shared variables are not guaranteed to be seen until the threads join
  - Full support for backtracking
- Concurrency:
  - `A &` – Sends out goal `A`, bindings in the variables of `A` (tells) will be exported to other workers sharing them
  - `A &&` – "Fair" version of the `&`/1 operator: if there is no idle worker, it creates one to execute `A` ("threads")
  - `wait(X)` – Suspends the execution until `X` is bound (also, `d_wait(X)`)
  - `ask(`$C$`)` – Suspends the execution until the constraint $C$ is satisfied
  - `A @ Id` – Placement operator: goal `A` is to be executed on worker `Id` (which may be remote).

# Using Parallelism: Examples

```
main :-
        read_input_list(L),
        collect_unloaded_hosts(Hosts),
        add_workers(Ids, Hosts),
        process_list(L),
        halt.

process_list([]).
process_list([H|T]) :-
        process(H) &
        process_list(T).

add_workers([Id|Ids],[Host|Hosts]) :-
        add_worker(Id,Host),
        add_workers(Ids,Hosts).
add_workers([],[]).
```

- Possible alternative using granularity control:

```
process_list([]).
process_list([H|T]) :-
    ( H < 5 ->
        process_list(T), process(H)
    ;   process(H) & process_list(T) ).
```

(CIAO parallelizing preprocessor: automatic parallelization and granularity control)

```
seq :-  move(red), move(green).
par :-  move(red) & move(green).
con :-  move(red) &, move(green).
thr :-  move(red) &&, move(green).
syn :-  sync_move(red, [ok|X]) &&, sync_move(green, X).

move(X) :-
        create(X,H),
        move(100, H),
        destroy(H).

move(0, _) :- !.
move(N, H) :- N > 0,
        move_step(H),
        N1 is N-1,
        move(N1, H).

sync_move(X, L) :-
        create(X,H),
        sync_move(100, H, L),
        destroy(H).

sync_move(0, _, _) :- !.
sync_move(N, H, L) :- N > 0,
        ask(L = [ok|R]),
        move_step(H),
        R = [ok|S],
        N1 is N-1,
        sync_move(N1, H, S).
```

# Using the CIAO Concurrency & Parallelism Operators


- `move(red), move(green).`
- `move(red) & move(green).`
- `add_worker(I), move(red) & move(green).`
- `move(red) &, move(green).`
- `delete_workers, move(red) &&, move(green).`
- `delete_workers, add_worker(alba,I), green @ I.`
- `move(red) & move(green).`
- `sync_move(red, [ok|X]) &&, sync_move(green, X).`
- `sync_move(red, [ok|X]) &, sync_move(green, X).`


CIAO_CANVAS #3
0@aguirre
CIAO_CANVAS #2
0@aguirre
CIAO_CANVAS #4
1@aguirre
CIAO_CANVAS #5
2@alba

# Some Implementation Issues

- Functionality implemented via a linda blackboard
- Goals and (all) solutions through blackboard
- Resulting scheduling similar to &-Prolog
- Shared variables implemented by attributed variables + blackboard:
  - variables involved in a parallel call are marked as a "communication" variable (i.e., shared)
  - done by attaching an attribute
  - communication variables are given unique identifiers
  - "shared" character is inherited during unification
  - standard tells done in place, tells to communication variables posted on the blackboard
  - asks do a blocking **`rd`** (read) on the blackboard
- All implementation done at source (Prolog) level (see ICLP'95 paper)
- Blackboard-based systems and shared variable communication-based systems – "different camps:" they can be easily unified using this technique!

# A Simple Producer/Consumer Program

```
go(L) :-
        consumer(L) &,
        producer(3,L).

producer(0,T) :- !, T = [].
producer(N,T) :- N > 0,
        T = [N|Ns],
        N1 is N-1,
        report(N,produced),
        producer(N1,Ns).

consumer(L) :-
        ask(L=[]), !.
consumer(L) :-
        ask(L=[H|T]),
        report(H,consumed),
        consumer(T).
```

... and the bindings posted to the blackboard

```
$b(0,[3|$shv(100)])
$b(100,[2|$shv(200)])
$b(200,[1|$shv(300)])
$b(300,[])
```

# Attributed Variables

- Attributed variables (AVs) [Huitouze '90] (and Meta-structures [Neumerkel '90]):
  - Proposed to implement constraint handling on top of a standard implementation (e.g. CLP(X) on Prolog).
  - Solvers implemented at the source level.
  - The ultimate example of the "glass box approach."
- Implementation [Holzbaur '92]:
  - Implemented a refined version of AVs in SICStus.
  - CLP($\Re$), CLP(Q)

  User accessible version of Carlsson's suspension variable (`freeze/2`).
- Also, ECL*i*PS*e* system (ECRC):
  - Useful performance/flexibility tradeoff.

# Performance

- "GHC in SICStus:"

| | attr | linda | attr/linda |
|---|---|---|---|
| Incomplete message protocol (30 messages) | | | |
| time | 17.5 | 6.4 | 2.7 |
| space | 3n+1 | 2n+1 | $\simeq$ 1.5 |
| oper. | 456 | 125 | 3.6 |
| Bounded buffer protocol (30 messages, 3 places) | | | |
| time | 21.0 | 6.6 | 3.2 |
| space | 6n-5 | 2n+4 | $\simeq$ 3 |
| oper. | 699 | 129 | 5.4 |
| One–many comm. w/acknowledge (20 messages, two readers) | | | |
| time | 25.1 | 4.6 | 5.45 |
| space | 10n+1 | 2n+1 | $\simeq$ 5 |
| oper. | 910 | 109 | 8.3 |

- "DDAS in &-Prolog:"

| Benchmark | 1 Proc. | 2 Proc. | 5 Proc. | 9 Proc. |
|---|---|---|---|---|
| qs-iap | 90 (1) | 50 (1.8) | 50 (1.8) | 50 (1.8) |
| qs-conc1 | 320 (1) | 170 (1.8) | 80 (4) | 60 (5.3) |
| qs-conc2 | 400 (1) | 210 (1.9) | 100 (4) | 80 (5) |
| nrev | 120 (1) | 80 (1.5) | 30 (4) | 30 (4) |

  - ◇ Some optimizations (e.g. no token passing).
  - ◇ Very naive implementation of kernel language.

# Conclusions

- Methods:
  - Blackboards.
  - Modeling and accessing information servers (active modules).
  - A simple distributed LP/CLP language using "worker teams".
  - Implementing distributed variable-based communication using attributed variables.
- Several useful concurrency and parallelism operators
- Attributed variables can be used for distributed communications
- All implementation can be done at source (Prolog) level
- Native support for concurrency in the underlying system can be very useful (e.g., in the CIAO run-time system, the &-Prolog abstract machine is used; similarly in BinProlog).
- AVs are very useful.
- Code provided as public domain standard libraries (`http://www.clip.dia.fi.upm.es`)

# Background

- Many proposed languages and systems (CLP(*), Prolog-IV, PNU-Prolog, Eclipse, Mercury, Gödel, λ-Prolog, Lygon, Babel, AKL, Oz,...; &-Prolog, Aurora, Muse, DDAS, Andorra-I, IDIOM, EAM, ...).
- They raise many questions for a future system. Examples:
  - From the language point of view:
    - higher-order/functions? native or as an add-on?
    - constraints? which systems? native or as an add-on?
    - low-level annotations for performance?
    - explicit or implicit concurrency?
    - concurrency by default?
    - sequential operator?
    - explicit or implicit search?
    - explicit or implicit distribution?
    - ...
  - From the compiler/abstract machine point of view:
    - which optimizations?
    - global analysis?
    - environment stacking or goal stacking?
    - which type of parallelism?
    - "multi sequential" model?
    - goal-based (delay) or continuation-based (cc) suspension?
    - interactions with non-determinism: encapsulation; copying/vs. sharing/vs. recomputation,
    - memory management/scheduling (+ also interactions w/non-determinism), ...

# Problem/Objectives

- Proposed systems are presented as a “bundled” package:
  language features,
  computation rule,
  parallelism principles,
  concurrency control, ...
- Can we study each issue separately?
- Can we reuse implementation technology?
  (analysis, optimizations, abstract machines, etc.)
- Vision [PPCP’94]:
  - ◇ Separation of issues
    (e.g., parallelism principle vs. control rules)
  - ◇ Identification of principles independently of level of granularity at which they are applied
    (e.g., goal level vs. binding level)

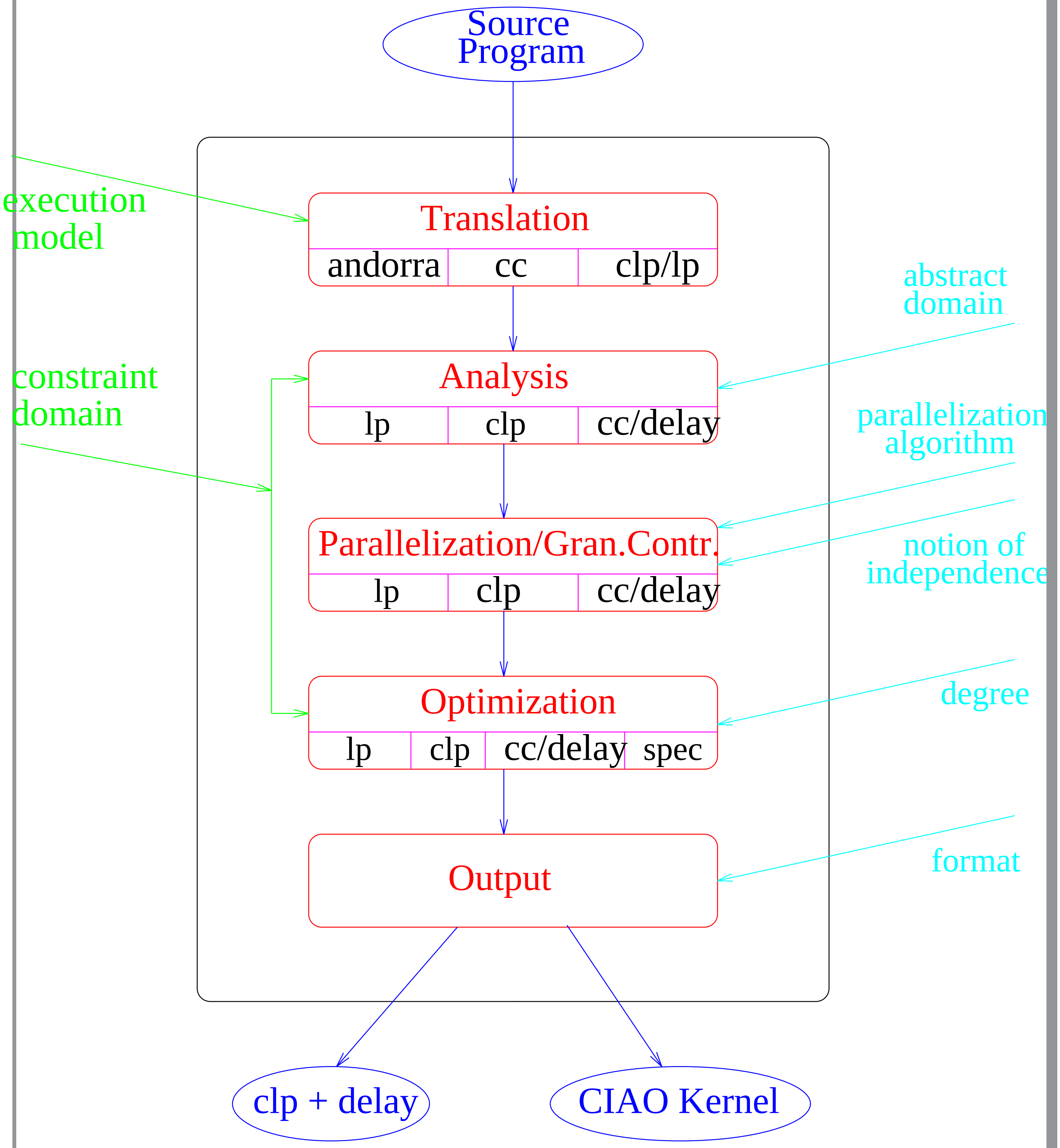
  together allow

  - ◇ identifying commonalities in languages and implementation technology,
  - ◇ finding unifying principles,
  - ◇ transferring ideas across systems,
  - ◇ implementing unified systems in a simple way.
- Objective: implement a system to test this viewpoint.

# The CIAO System

- Functionality:
  - Multi-paradigm compiler / programming environment.
  - Supports the programming styles of LP (Prolog), CLP (constraints), CC (concurrency), Andorra,...
    (+ functions, higher order syntax support, active objects, scripts, records, WWW interface, ...)
  - Concurrency, parallelism, and distributed execution.
  - Largely "backwards compatible" with trad. LP/CLP systems.
- Implementation approach:
  - Much functionality implemented by compilation to a simple kernel language via source to source transformations.
  - Performs global analysis.
  - Compile-time optimizations via program transformation, incl. multiple (abstract) specialization, parallelization, ...
- Can be used to compile (with different efficiency) to different implementation platforms:
  - General purpose LP/CLP systems with delay primitives and attributed variables (e.g., SICStus, Eclipse)
  - The CIAO abstract machine (an enhancement of the &-Prolog abstract machine)
- Compiler writer's workbench: Driver / very flexible experimentation platform.
- Also very useful system/library for developing applications!

# The CIAO Compiler




```
:- module(module_name, entry_points, language(constr_system))
```

# Example

```
% :- module(fib,[fib/2],ciao(h)).
:- module(fib,[fib/2],andorra(h)).
:- entry(fib(g,f)).
fib(0,1).
fib(1,1).
fib(A,B) :-
        A>1,
        C is A-1,
        D is A-2,
        fib(C,E),
        fib(D,F),
        B is F+E.
```

# CIAO and Other CC Systems

- Input from other LP/CC systems:
  - CC: entailment-based synchronization.
  - NU-Prolog/Par NU-Prolog: transformation to delay declaration for support of CIAO on conventional systems.
  - AKL: encapsulation.
  - OZ: modules, applications of records.
  - Shared with QE-Janus: "quiche eating" implementation approach.
- Main differences:
  - "Sequential by default" vs. "concurrent by default."
  - Explicit concurrency (and parallelism) operators ("threads").
  - Distributed implementation.
  - Extensive global analysis and optimization (e.g., automatic static parallelization, suspension reduction).
  - Designed to be portable to conventional LP/CLP systems.
- Other issues:
  - Active modules.
  - WWW interface.
  - Functions, HO, scripts, ...

# Compiler Issues: Transformations/Optimizations

- Examples of transformations/techniques used:
  - Supporting CLP via attributed variables.
  - Distributed execution on standard CLP/LP.
  - Supporting CC on standard CLP/LP systems (with delay).
  - Supporting the Andorra model in CLP/LP systems.
  - Functions/higher order.
- Analyses used / characteristics:
  - Top-down framework with efficient dynamic fixpoint (PLAI).
  - Modes, types, sharing (aliasing), independence, etc.
  - Several domains over Herbrand: *SH*, *SH+FR*, *ASub*, *SH+ASub*, *SH+FR+ASub*, *Path*, *Types*, ...
  - Over constraints: *Def*, *Fr*, *FD*, *LSign*, *DiffLSign*, ...
  - Support for dynamic scheduling (concurrency).
  - Support for incremental analysis.
  - Support for full languages (e.g., ISO-prolog).
  - Cost analysis (upper and lower bounds).
- Examples of optimizations performed:
  - Compile-time elim. of run-time tests via (abstract) PE.
  - Multiple (abstract) specialization (e.g., loop invariants).
  - LP/CLP/CC parallelization.
  - Optim. of synchronization / sched. anal. (for delays and CC).
  - Goal and constraint reordering (optimization of search).
  - Granularity control.

# Language Visions?

- On the future LP Language: kernel CIAO some interesting ideas?
  - Backwards compatible with LP/CLP.
  - Can use existing implementation technology.
  - Incorporates some language solutions:
    - Separation of control rules (e.g., Andorra) from parallelism and optimizations.
    - Integration of several in the same framework.
    - Sequential operator.
    - Separation of parallelism and concurrency.
    - Explicit request for fairness.
    - Distribution primitives.
    - Active modules/objects.
    - ...
  - Final thoughts – minor things matter, e.g., in CIAO:
    - tcl/tk interface.
    - Stand-alone executables, linkables, *and scripts*.
    - Small executables.
    - *html* interface.
    - ...