An Overview of Ciao
(and its uses of DataLog in Program Analysis/Optimization)

Manuel Hermenegildo$^{4,1}$
(with F. Bueno,$^1$ G. Puebla,$^1$ M. Carro,$^1$ P. López-García,$^1$ J. Morales,$^2$ E. Mera,$^1$ P. Chico,$^1$ J. Navas,$^3$ R. Haemmerlé,$^1$ M. Méndez,$^3$ A. Casas,$^3$ J. Correas,$^2$ E. Albert,$^2$ P. Arenas,$^2$ )

The Ciao Development Team

$^1$CS Dept., T.U. of Madrid
$^2$CS Dept., Complutense U. of Madrid
$^3$CS and EECE Depts., U. of New Mexico, USA
$^4$IMDEA Software Research Institute

Datalog 2.0 – Oxford, March 19, 2010
Motivation and Approach

Objectives:

Next-generation, high-level, *multiparadigm* programming language: Ciao.

*Program development environments which perform, as part of compilation:*

- Verification / debugging
  (i.e., detect bugs and offer guarantees of safety, reliability, and efficiency.)
- Optimization (optimized compilation, parallelization, ...).

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

- Apply in a real system, with users –reality check!
- Support also mainstream languages (e.g., Java / Java bytecode).

Several uses of Datalog and related techniques.
Ciao Packages and Paradigms

- **Built in layers over a small, LP-based *kernel***:
  - “Packages” provide *syntactic and semantic extensions and restrictions* on a per-module basis.

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

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

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

- **Objects**.

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

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

+ Many other packages: type systems, records, *PiLLoW, RDF, XPath, DSLs*, ...
Some Uses of Datalog and Related Technology

- Intermediate representation for several Ciao analyses.
  - Groundness (mode) analysis: `def`, BDDs.
  - Definiteness propagation in analysis of (C)LP.
  - Assertion checking: comparator.
  - VC generation / checking in Abstraction Carrying Code.
  - Simplification of parallelization conditions.

- Also for Java: nullity, aliasing / sharing, resource usage.

- Simple solvers based generally on tabling.

- Bottom-up fixpoint evaluators also used in, e.g., shape / type inference.
  - Uses magic sets, etc.

- ASP modules (ASP-Prolog, Pontelli et al.).

+ All tools written in Ciao (= Datalog++++ :-)).
Program Analysis and Optimization in Ciao

- Compute Safe over- and/or under-approximations of program semantics,
  \([P]_{\alpha^+}\) or \([P]_{\alpha^-}\)
  generally based on modular, polyvariant abstract interpretation:
  \[
  \forall x \in D : \gamma(\alpha(x)) \supseteq x, \text{ and }
  \forall y \in D_\alpha : \alpha(\gamma(y)) = y.
  \]
  \[
  \text{lfp}(S^\alpha_P) = [P]_\alpha \supseteq \alpha([P]) \text{ terminates.}
  \]

- Apply such approximations to verification, optimization.
- Domains: types, modes, pointer sharing, cost, sizes, termination, determinacy, non-fail, ...
Program verification/diagnosis: compare $[P]$ with intended semantics $\mathcal{I}$ e.g.:

<table>
<thead>
<tr>
<th>Property</th>
<th>Definition</th>
<th>Sufficient condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P$ is partially correct w.r.t. $\mathcal{I}$ if</td>
<td>$\alpha([P]) \subseteq \mathcal{I}_\alpha$</td>
<td>$[P]<em>{\alpha^+} \subseteq \mathcal{I}</em>\alpha$ if</td>
</tr>
<tr>
<td>$P$ is complete w.r.t. $\mathcal{I}$ if</td>
<td>$\mathcal{I}_\alpha \subseteq \alpha([P])$</td>
<td>$\mathcal{I}<em>\alpha \subseteq [P]</em>{\alpha^+}$</td>
</tr>
<tr>
<td>$P$ is incorrect w.r.t. $\mathcal{I}$ if</td>
<td>$\alpha([P]) \not\subseteq \mathcal{I}_\alpha$</td>
<td>$[P]<em>{\alpha^+} \not\subseteq \mathcal{I}</em>\alpha$, or</td>
</tr>
<tr>
<td></td>
<td>$\mathcal{I}_\alpha \not\subseteq \alpha([P])$</td>
<td>$[P]<em>{\alpha^+} \cap \mathcal{I}</em>\alpha = \emptyset \land [P]_{\alpha^+} \not\subseteq \emptyset$</td>
</tr>
<tr>
<td>$P$ is incomplete w.r.t. $\mathcal{I}$ if</td>
<td>$\mathcal{I}_\alpha \not\subseteq \alpha([P])$</td>
<td>$\mathcal{I}<em>\alpha \not\subseteq [P]</em>{\alpha^+}$</td>
</tr>
</tbody>
</table>

Usually, only partial descriptions of $\mathcal{I}$ are available, typically as assertions.

**Problem:** difficulty in computing $[P] \rightarrow$ use *abstract interpretation* to compute a safe approximation $[P]_{\alpha^+}$. $[P]_{\alpha^+}$ indicates $[P]_{\alpha} \supseteq \alpha([P])$.

Specially attractive if compiler computes (most of) $[P]_{\alpha^+}$ anyway.
$I_\alpha$ (partial spec.) provided via a language of optional assertions.

- State properties at relevant point (pre, post, global, pp).
- Talk about “properties,” predefined or user-defined (in the source language).
- Types, modes, pointer sharing, cost, sizes, termination, determinacy, non-fail, ...
$I_\alpha$ (partial spec.) provided via a language of optional assertions.

- State properties at relevant point (pre, post, global, pp).
- Talk about “properties,” predefined or user-defined (in the source language).
- Types, modes, pointer sharing, cost, sizes, termination, determinacy, non-fail, ...
Assertion Language: Properties

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

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

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

Types are a special case of property (e.g., regtypes).
But also, e.g., argument sizes, instantiation states, sizes, cost, ...

- Arbitrary predicates in restricted logic (a subset of Ciao).
- Some conditions on them: termination, no instantiation, ...
- Many predefined in system libs, some of them “native” to an analyzer.
- Can also be 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).
Assertion Language: *Pred* Assertions

\[
\text{:- pred}\ PredPattern\ [\ :\ Pre\ ]\ [\ =>\ Post\ ]\ [\ +\ Comp]\ .
\]

- Closed on calls: cover all uses of a predicate (they imply a calls assertion).
- Assertion status: check, true/false, trust, checked.

Some examples, and some syntactic sugar:

- \text{:- pred qsort(X,Y) => sorted(Y).}
- \text{:- pred qsort(X,Y) : list(int) * var => sorted(Y) + (is_det,not_fails).}
- \text{:- pred qsort(X,Y) : var * list(int)) => ground(X) + not_fails.}
- \text{:- pred foo(X,Y) : ground * var => (ground(Y), X>Y) + det.}
- \text{:- pred foo(X,Y) : var * ground => (ground(X), X>Y).}
- \text{:- trust pred is/2 => num * numexpr.}
- \text{:- modedef +X : nonvar(X).}
- \text{:- pred sortints(+L,-SL) :: list(int) * list(int) => sorted(SL)
  # "@var{SL} has same elements as @var{L}.".}
The Abstraction Carrying Code (ACC) Scheme  [COCV04,LPAR04]

$$[P]_\alpha = \text{Analysis} = \text{lfp}(\text{analysis}\_\text{step})$$  
Certificate $\subset [P]_\alpha$  
Checker = $\text{analysis}\_\text{step}$

Scheme incorporated in CiaoPP, with domains: types, modes, data structure shape (including pointer sharing), bounds on data structure sizes, determinacy, termination, non-failure, bounds on resource consumption (time or space cost), ...
Big discussion (90’s :-))): comparison with “classical” Types

- Allows going well beyond the “straight-jacket” of classical type systems:

<table>
<thead>
<tr>
<th>“Traditional” Types</th>
<th>CiaoPP Assertion-based Model [AADEBUG’97]</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Property” language limited by decidability</td>
<td>Much more general property language</td>
</tr>
<tr>
<td>May need to limit programming language</td>
<td>No need to limit programming language</td>
</tr>
<tr>
<td>“Untypable” programs rejected</td>
<td>Run-time checks introduced</td>
</tr>
<tr>
<td>(Almost) Decidable</td>
<td>Decidable and Undecidable (and Approximated)</td>
</tr>
<tr>
<td>Expressed in a special language</td>
<td>(Expressed in the source language –for LP)</td>
</tr>
<tr>
<td>Types must be defined</td>
<td>Types can be defined or inferred</td>
</tr>
<tr>
<td>Assertions are only of type “check”</td>
<td>“check”, “trust”, ...</td>
</tr>
<tr>
<td>Type signatures and assertions different</td>
<td>Type signatures are assertions</td>
</tr>
</tbody>
</table>

...without giving up much (types are included as just another kind of property).

- Some key issues:
  - Approximation
  - Abstract Interpretation

Suitable assertion language
Powerful abstract domains

- Worst best when properties and assertions can be expressed in the source language (i.e., the source language supports predicates and constraints).
Also, optimizations

- Preprocessor architecture useful not just for verification / debugging, but also for optimization:
  - Source-level optimizations:
    - Partial evaluation, (multiple) (abstract) specialization, slicing, ...
  - Low-level (WAM) optimizations:
    - Use of specialized instructions.
    - Optimized native code generation.
  - Parallelization.
Automatic Program Parallelization

- Parallelization process starts with dependency graph:
  - edges exist if there can be a dependency,
  - conditions label edges if the dependency can be removed.

- Annotation: encoding of parallelism in the target parallel language:
  
  \[ g_1(\ldots), g_2(\ldots), g_3(\ldots) \]

- Global *sharing/aliasing* analysis: reduce/eliminate checks in conditions.

- Granularity control: based on cost / size analysis.
Automatic Program Parallelization (Contd.)

Example:

\[
\text{qs([X|L],R) :- part(L,X,L1,L2),}
\]
\[
\text{ qs(L2,R2), qs(L1,R1),}
\]
\[
\text{ app(R1,[X|R2],R).}
\]

Might be annotated in &-Prolog (or Ciao Prolog), using local analysis, as:

\[
\text{qs([X|L],R) :-}
\]
\[
\text{ part(L,X,L1,L2),}
\]
\[
\text{ ( indep(L1,L2) ->}
\]
\[
\text{ qs(L2,R2) & qs(L1,R1)}
\]
\[
\text{ ; qs(L2,R2), qs(L1,R1) ),}
\]
\[
\text{ app(R1,[X|R2],R).}
\]

Global analysis would eliminate the \text{indep(L1,L2)} check.
Other Related Recent Work

- Framework adapted to Java and Java bytecode (Mobius):
  - Developed specific framework for Java/Java bytecode.
  - Analysis/validation of Java bytecode via P.Eval. of interpreter.
- Scalability, modularity.
- ACC:
  - Reduced certificates.
  - Incremental ACC (and other advanced PCC scenarios).
- Extension of cost analysis to *time* bounds.
- Cost analysis of Java bytecode
- Extension to generic user-defined resources (Prolog and Java).
  - Examples: bytes sent over Internet, files open, DB accesses, etc.
- Abstract diagnosis.

(+ IMDEA-Software Development Technology Institute!)
Some Members of The Ciao Forge

- Ciao is really a widely distributed collaborative effort:
  - Directly within the CLIP Group:
  - Plus lots of contributors worldwide:
    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. Deransart (INRIA), J. Gallagher (Roskilde University), C. Holzbauer (Austrian Research Institute for AI), M. Codish (Beer-Sheva), SICS, ...
Some Selected Bibliography on the Ciao System

All papers can be found on line at: [http://clip.dia.fi.upm.es/clippubsbyyear](http://clip.dia.fi.upm.es/clippubsbyyear) and [http://clip.dia.fi.upm.es/clippubsbytopic](http://clip.dia.fi.upm.es/clippubsbytopic)

**System manual:**


**Overall design and philosophy:**


Functions, higher order, lazyness:


Tabling:


Objects:


Auto-documenter:

Asbtract machine and low-level optimization:


Automatic parallelization:


Cost analysis and granularity control in parallelism:


The overall program development framework (CiaoPP):


Abstraction carrying code:


Partial evaluation:


**Scalability, modularity of analysis, debugging, and verification:**


**Some applications of the CiaoPP framework to Java bytecode:**


