A Partial Deducer Assisted by Predefined Assertions and a Backwards Analyzer

Elvira Albert*, Germán Puebla and John Gallagher*****

(*) *Complutense University of Madrid (Spain)*
(**) *Technical University of Madrid (Spain)*
(***) *University of Roskilde (Denmark)*

5th International Workshop on the Implementation of Logics (WIL'04)

Uruguay, March 13, 2005

Introduction: The Ciao Program Development System

- Ciao is a next-generation (C)LP programming environment – features:
  - Public domain (GNU license).
  - Pure kernel (*no "built-ins"*); subsumes ISO-Prolog (transparently) via *library*.
  - Designed to be `extensible` and `analyzable`.
  - 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

- Components of the environment (independent):

`ciaosh`:	Standard top-level shell.
`ciaoc`:	Standalone compiler.
`ciaosi`:	Script interpreter.
`lpdoc`:	Documentation Generator (info, ps, pdf, html, ...).
`ciaopp`:	Preprocessor.

  + Many libraries:
    - Records (argument names).
    - Persistent predicates.
    - Transparent interface to databases.
    - Interfaces to C, Java, tcl-tk, etc.
    - Distributed execution.
    - Internet (PiLLoW: HTML, VRML, forms, http protocol, etc.), ...

The Ciao Preprocessor

- A standalone preprocessor to the standard clause-level compiler.
- Performs source-to-source transformations:
  - Input: logic program (optionally w/assertions & syntactic extensions).
  - Output: *error/warning messages* + *transformed logic program*, with
    - ★ Results of analysis (as assertions).
    - ★ Results of static checking of assertions.
    - ★ Assertion run-time checking code.
    - ★ Optimizations (specialization, partial evaluation, etc.)
- By design, a generic tool – can be applied to other systems (e.g., CHIP $\longrightarrow$ CHIPRE).
- Underlying technology:
  - Modular polyvariant abstract interpretation.
  - Modular abstract multiple specialization.

Applications of Abstract Interpretation

- A number of applications of Abstract Interpretation of (C)LP:
  - Inference of complex properties of programs.
  - Program debugging.
  - Program validation.
  - Program optimization (e.g., partial evaluation, specialization, parallelization).
  - Program documentation.
- Some practical issues:
  - The *assertion* language.
  - Dealing with built-ins and complex language features.
  - Modular analysis (including libraries).
  - Efficiency and incremental analysis (only reanalyze what is needed).

Properties and Assertions

- Assertion language suitable for *multiple purposes*.
- Assertions are typically *optional*.
- Properties (include *types* as a special case):
  - Arbitrary predicates, (generally) *written in the source language*.
  - Some predefined in system, some of them "native" to an analyzer.
  - Others user-defined.
  - Should be "runnable" (but property may be an approximation itself).

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

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

```
:- typedef list::= [];[_|list].
```

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

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

Properties and Assertions

- Basic assertions:

```
:- success   PredDesc   [   : PreC ]   => PostC .
:- calls     PredDesc       : PreC .
:- comp      PredDesc   [   : PreC ]   + CompProps .
```

Examples:

```
:- success qsort(A,B) : list(A) => ground(B).
:- calls qsort(A,B) : (list(A),var(B)).
:- comp qsort(A,B) : (list(A,int),var(B)) + (det,succeeds).
```

- Compound assertion (syntactic sugar):

```
:- pred   PredDesc   [   : PreC ]   [ => PostC ] [ + Comp ] .
```

Examples:

```
:- pred qsort(A,B) : (list(A,int),var(B)) => sorted(B) + (det,succeeds).
:- pred qsort(A,B) : (var(A),list(B,int)) => ground(A) + succeeds.
```

Properties and Assertions

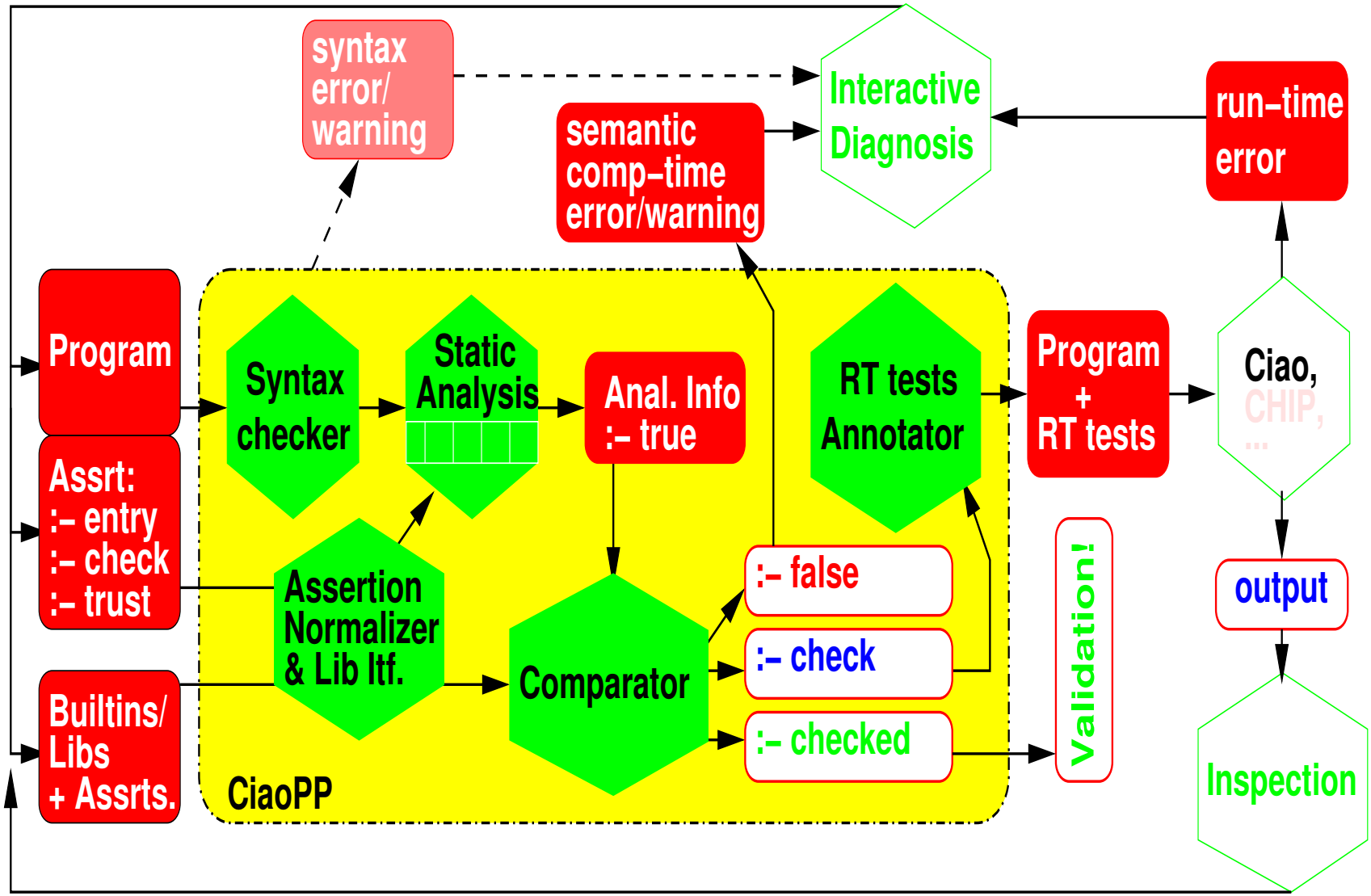
- Assertion *status*:
  - `check` (default) – intended semantics, to be checked.
  - `true`, `false` – actual semantics, output from compiler.
  - `trust` – actual semantics, input from user (guiding compiler).
  - `checked` – validation: a `check` that has been proved (same as a `true`).

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

Analysis

- `ciaopp` includes some basic analyzers:
  - The PLAI generic, top-down analysis framework.
    - ★ Several domains: modes (ground, free), independence, patterns, etc.
    - ★ Incremental analysis, analysis of programs with delay, ...
  - Gallagher's backwards analysis.
    - ★ Captures dependencies by program transformation.
  - Advanced analyzers (GraCos/CASLOG) for complex properties: non-failure, coverage, determinism, sizes, cost, ...
- Issues:
  - Reporting the results $\longrightarrow$ "true" assertions.
  - Helping the analyzer $\longrightarrow$ "entry/trust" assertions.
  - Dealing with builtins $\longrightarrow$ "trust" assertions.
  - Incomplete programs $\longrightarrow$ "trust" assertions.
  - Modular programs $\longrightarrow$ "trust" assertions, interface (`.itf`, `.asr`) files.
  - Multivariance, incrementality, ...

Integrated Validation/Diagnosis in the Ciao Preprocessor



Using Analysis Results in Program Optimization

- Eliminating run-time work at compile-time.
  - Low-level optimization.
  - Partial evaluation
    Specialize programs w.r.t. known input data
  - Abstract multiple specialization.
    Ditto on (possibly) multiple versions of each predicate.
- Automatic program parallelization:
  strict and non-strict Independent And-Parallelism.
- Automatic task granularity control.
- Optimization of other control rules / languages (e.g., Andorra).

Partial Deduction

- Partial Deduction (PD) specializes a program w.r.t. part of its known input data (program specialization)
- Given an input program and a set of atoms, PD algorithm applies an *unfolding rule* in order to compute finite (possibly incomplete) SLD trees for atoms.
  - *computation rule*: given a goal $\leftarrow A_1, \ldots, A_R, \ldots, A_k$ determines the selected atom A_R
  - *profitability test:* decides whether unfolding (or evaluation) of A_R is *profitable*
- This process returns a set of *resultants* (or residual rules), i.e., a residual program, associated to the root-to-leaf derivations of these trees.

Non-leftmost Unfolding

- *Non-leftmost* unfolding is essential in partial deduction in some cases for the satisfactory propagation of static information.
- Given a goal $\leftarrow A_1, \ldots, A_n$, it can happen that the profitable criterion does not hold for the leftmost atom A_1.
  - A_1 is an atom for an internal predicate and, 1) unfolding A_1 endangers termination or 2) the atom A_1 unifies with several clause heads
  - A_1 is an atom for an external predicate, it can happen that A_1 is not sufficiently instantiated so as to be executed at this moment.
- Thus, it may be profitable to unfold non-leftmost atoms.
- Computation rule which is able to detect the above circumstances and "jump over" atoms whose profitability criterion is not satisfied.
- Proceed with the specialization of another atom in the goal as long as it is correct.

Impure Predicates

- For pure logic programs without builtins, non-leftmost unfolding is safe thanks to the independence of the computation rule
- Non-leftmost unfolding poses problems in the context of *full* Prolog programs with *impure* predicates (independence does not hold).
- `var/1` is an *impure* atom (the goal `var(X),X=a` succeeds with computed answer X/a whereas `X=a,var(X)` fails)
- *Backpropagation of bindings:* Given the goal $\leftarrow$ `var(X),X=a`, if we allow the non-leftmost unfolding step which binds the variable X, the goal will fail, either at specialization time or at run-time, whereas the initial goal succeeds in LD resolution.
- Solution: represent explicitly the bindings by using unification rather than backpropagating them (applying them onto leftmost atoms).
- This guarantees correctness, but introduces some inaccuracy (bindings are hidden from atoms to the left of the selected one).

Assertions about Purity

- Allow unfolding non-leftmost atoms by classifying predicates into pure and impure (avoiding backpropagation in impure predicates).
- Simple reachability analysis: as soon as an impure predicate p can be reached from a predicate q, also q is considered impure.
- Our work improves on existing techniques by providing a more refined notion of impurity: we define purity at the level of individual atoms.
- $var(X)$ is impure (binding s), whereas the atom $var(f(X))$ is not (it is no longer binding sensitive). We *reduce* substantially the situations in which backpropagation has to be avoided.
- Assertions state sufficient conditions under which atoms are pure. Impurities: binding-sensitive, errors and side effects.

```
:- trust comp p(X1,...,Xn) : SC + bindins.
:- trust comp p(X1,...,Xn) : SC + error_free.
:- trust comp p(X1,...,Xn) + sideff.
```

Backwards Analysis

- Recent developments in backwards analysis of logic program have pointed out novel applications in termination analysis and inference of call patterns which are guaranteed not to produce any runtime error.
- We propose a new application of backwards analysis for automatically inferring binding insensitive, error free and side-effect free annotations which are useful to this purpose.
- Automatically figuring out when a substitution can be safely backpropagated onto a call whose execution reaches an impure predicate
- The analyzer starts from a program and an initial set of assertions which state the properties of interest for external predicates.
- The analysis algorithm propagates this information backwards in order to get the appropriate assertions for all predicates.

Automatic Inference of Assertions by Backwards Analysis

- Predefined assertions in `Ciao` for predicate `ground/1`:

```
:- trust comp ground(X) : true + error_free.
:- trust comp ground(X) + sideff_free.
:- trust comp ground(X) : ground(X) + bind_ins.
```

- The `Ciao` program:

```
:- module(main_prog,[main/2],[]).
:- use_module(comp,[long_comp/2],[]).
main(X,Y)    :- problem(X,Y), q(X).
problem(a,Y):- ground(Y),long_comp(a,Y).
problem(b,Y):- ground(Y),long_comp(b,Y).
q(a).
```

Automatic Inference of Assertions by Backwards Analysis

- Predicate `long_comp/2` is externally defined in module `comp` where also these predefined assertions for it are:

```
:- trust comp long_comp(X,Y) : true + error_free.
:- trust comp long_comp(X,Y) + sideff_free.
:- trust comp long_comp(X,Y) : ground(Y) + bind_ins
```

- From the program and the available assertions (for `long_comp/2` and `ground/1`), the backwards analyzer infers the following assertions for `problem/2`:

```
:- trust comp problem(X,Y) : true + error_free.
:- trust comp problem(X,Y) + sideff_free.
:- trust comp problem(X,Y) : ground(Y) + bind_ins.
```

The last assertion indicates that calls performed to `problem(X,Y)` with the second argument being ground are not binding sensitive. This will be very useful information for the specializer.

Automatic Inference of Assertions by Backwards Analysis

- Consider a deterministic unfolding rule and the entry goal: " `:- entry main(X,a).`"
- The unfolding rule performs an initial step and derives the goal `problem(X,a),q(X)`.
- Now, it cannot select the atom `problem(X,a)` because its execution performs a non deterministic step.
- Fortunately, the assertions inferred for `problem(X,Y)` allow us to jump over this atom and specialize first `q(X)`.
- Then `X` gets instantiated to `a` and the unfolding rule already can select the deterministic atom `problem(a,a)` and obtain the specialized fact " `main(a,a).`"
- Otherwise, the specialized program would:

```
main(X,a):-problem(X,a),q(X).
```

which is much less efficient than our specialization since the execution of the call to `long_comp` remains residual

Conclusions

- Motivated by recent developments in the *backwards* analysis of logic programs, we propose a partial deduction algorithm which can handle impure features and non-leftmost unfolding in a more accurate way.
- We outline by means of examples some optimizations which are not feasible using existing partial deduction techniques.
- We argue that our proposal goes beyond existing ones and is:
  1. accurate, since the classification of pure vs impure is done at the level of atoms instead of predicates,
  2. extensible, as the information about purity can be added to programs using assertions which can guide the partial deduction process, and
  3. automatic, since backwards analysis can be used to automatically infer the required assertions.

  Our approach has been implemented in the context of `CiaoPP`, the abstract interpretation-based preprocessor of the `Ciao` logic programming system.