Programming with Global Analysis

M. Hermenegildo

in collaboration with

G. Puebla, F. Bueno, P. Deransart
W. Drabent, G. Ferrand, J. Małuszyński

1 Technical University of Madrid
2 INRIA-Rocquencourt
3 Polish Academy of Sciences
4 University of Orléans
5 Linköping University

(Partially Supported by ESPRIT Project DiSCiPI: Debugging Systems for Constraint Programming.)
Introduction

- Much progress made in global program analysis, generally based on *abstract interpretation*.
- Current results/systems allow:
  - Inferring non-trivial information
    (from types and modes to cost, termination, non-failure...)
  - Dealing with full languages and large (modularized) programs.
- Results have been applied generally to program transformation/optimization.
- Tutorial objective: explore applications in *program development*.
  - In particular, in *validation* and *diagnosis*.
Validation and Diagnosis

- Users expect programs to satisfy certain requirements (specifications).
  - proved to hold (i.e., they are verified),
- Requirements may be:
  - or to be violated (i.e., a symptom is detected).
- Diagnosis: identifying the part of the program responsible for the violation.
- The traditional approach generally uses two (separate) mechanisms, e.g.:
  - Run-time testing:
  - Proof-based validation (generally, non-automated).
    (Exception: types.)
Actual and Intended Semantics

- Semantics associate a *meaning* to a given syntax.
- A meaning is an element of a *semantic domain* ($D$).
- Semantics allow:
  - capturing the characteristics (observables) of interest,
  - while hiding others which are not relevant.
- We consider the class of *fixpoint semantics*, using *sets*:
  - *Actual semantics* of a program $P$ is $[P]$ and corresponds to a set.
  - $[P]$ is the *least fixpoint* of a semantic operator $S_P$.
- Program validation and diagnosis both compare actual semantics $[P]$ of the program with an intended semantics $\mathcal{I}$ for the same program.
- $\mathcal{I}$ can be seen as the semantics of an intended program
  (which does not exist in general).
- Usually, only partial descriptions of $\mathcal{I}$ are available.
Program Validation Tasks

- Set theoretic definition of validation tasks:
  - $P$ is partially correct w.r.t. $I$ iff $[P] \subseteq I$.
  - $P$ is complete w.r.t. $I$ iff $I \subseteq [P]$.
  - $P$ is incorrect w.r.t. $I$ iff $[P] \not\subseteq I$.
  - $P$ is incomplete w.r.t. $I$ iff $I \not\subseteq [P]$.

- Incorrectness and incompleteness indicate that diagnosis should be performed.

- **Problem**: difficulty in computing $[P]$.

- **Possible solutions / alternatives**:
  - Proving sufficient conditions (as in *diagnosis by proof*).
  - Program testing, run-time checking, manual tracing, etc. (but incomplete and generally expensive)
  - *Approximating* $[P]$ directly (safely)
    (approximate or *abstract validation* [Bourdoncle93], [ESOP’96], ...).
Approximating the Intended Semantics

- Using the exact intended semantics for validation and diagnosis is in general also not realistic because $\mathcal{I}$ may be:
  - only partially known,
  - infinite,
  - too expensive to manipulate, ...

- We consider three types of approximations:
  - Superset ($A^+$): $A \subseteq A^+$.
  - Subset ($A^-$): $A^- \subseteq A$.
  - Existential ($A^!$): $A^! \cap A \neq \emptyset$.

- For example, in [DNTM89] the approximations used were of types $\mathcal{I}^-$, $(\overline{\mathcal{I}})^!, \mathcal{I}^!$ and $\overline{\mathcal{I}}^+$ or, equivalently, $(\overline{\mathcal{I}})^-$. 
Examples of Intended Semantics

- Given:
  
  \[
  \begin{align*}
  \text{sorted}([\_]). \\
  \text{sorted}([\_]). \\
  \text{sorted}([X,Y|Z]) & : \ X > Y, \ \text{sorted}([Y|Z]). \\
  \end{align*}
  \]

- A subset of the programmer’s intentions:
  
  \[
  \mathcal{I}_1 = \{ sorted([X]) \mid X \text{ is an integer} \}
  \]

- A superset of the programmer’s intentions:
  
  \[
  \mathcal{I}_2 = \{ sorted(L) \mid L \text{ is a list of integers} \}
  \]
Assertion Languages

- Assertion: expression which identifies an element of $D$.
- Depends on the semantic domain $D$ (in our case, it has to describe sets).
- Examples of possible assertion languages:
  - Any language that can describe sets (e.g., formulas of first order predicate calculus).
  - A specialized language over an “abstract domain” (e.g., types).
  - The source language – some examples:
    - Prolog programs used as assertions in [DNTM89].
    - Prolog programs used as types and properties in the CIAO system [ESOP’96].
      * Specially useful in higher-level logic languages (e.g., LP, CLP).
      * The properties in the assertions can be used as run-time tests.
Assertions (and Properties)

- Multiple uses/roles:
  - Run-time checking (e.g., pre/post-cond) – general properties, "check".
  - Compile-time checking (e.g., types) – decidable, compulsory, "check".
  - Replacing the oracle – general declarative properties, "check".
  - Providing info to optimizer (e.g., pragmas) – general properties, "trust".
  - General comm. w/compiler (e.g., entry, trust) – general properties, "trust".

- Objectives:
  - Propose an assertion language suitable for all these purposes.
    (When possible, keep backwards compatibility w/ISO & popular platforms.)
  - Study formally interaction with different debugging tools (and implement!).

- Important issue: whole system should deal safely with general, undecidable properties, and incomplete information → safe approximations.

- Different program development tools may use different parts of the language.
An Assertion Language: Properties

- Arbitrary predicates, (generally) written in the source language.
- Some predefined in system, others user-defined.
- Should terminate (but code may be an approximation of the property).
- Types are just a special case (example: regular programs).
- Some examples (all examples are in CIAO assertion syntax – see [ILPS’97 WS]):

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

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

:- type integer/1.

:- type peano_int/1.

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

% is built-in

:- type peano_int/1.
peano_int(0).
peano_int(s(X)) :- peano_int(X).
```
Declarative Assertions: *Superset* and *Subset*

- Written by the user, optional — they describe the intended semantics ($I$).
- Two kinds:
  - **“Superset” assertions:**
    
    ```prolog
    :- inmodel Pred => Props.
    
    * Describe (a superset of) the results of a predicate.
    * Used for correctness debugging (violation is a symptom).
    * Example:
      
      ```prolog
      :- inmodel qsort(A,B) => list(B).
    ```

  - **“Subset” assertions:**
    
    ```prolog
    :- inmodel Pred <= Props.
    
    * Describe (a subset of) the results of a predicate.
    * Used for completeness debugging (violation is a symptom).
    * Example:
      
      ```prolog
      :- inmodel qsort(A,B) <= (A=[2,1],B=[1,2]).
    ```
Basic Operational Predicate Assertions: *Success*

- Written by the user, optional — describe intended (operational) semantics \((\mathcal{I})\).
- All “superset.”
- Properties *should* apply to all run-time invocations of a predicate.
- “*Success*” assertions:
  
  \[\text{:- success } Pred \Rightarrow PostCond.\]

  - Describe *post-conditions* of a predicate.
  - Any declarative superset assertion is also a success assertion (due to correctness of the operational semantics).
  - Example:
    \[\text{:- success } \text{qsort}(A,B) \Rightarrow \text{list}(B).\]

- Restricting to a subset of calls:
  
  \[\text{:- success } Pred : PreCond \Rightarrow PostCond.\]

  - Example:
    \[\text{:- success } \text{qsort}(A,B) : \text{list}(A) \Rightarrow \text{list}(B).\]
Basic Operational Predicate Assertions: *Calls* and *Comp*

- **“Calls”** assertions:

  ```prolog
  :- calls Pred : Props.
  ```

  - Describe properties of the calls to a predicate.
  - Example:

    ```prolog
    :- calls qsrt(A,B) : (list(A),var(B)).
    ```

- **“Comp”** assertions:

  ```prolog
  :- comp Pred : PreCond + CompProps.
  ```

  - Describe props of predicate execution (determinacy, non-failure, cost, ...).
  - Example:

    ```prolog
    :- comp qsrt(A,B) : (intlist(A),var(B)) + (det,no_fail).
    ```

- Most general, but others always preferred if possible.
Compound Operational Predicate Assertions: *Pred* Assertions

- Issues in practice with previous assertions:
  - Verbose in some cases: more compact notation desired.
  - Closedness of calls in multiple success assertions.
- "*Pred*" assertions:

  \[
  \text{:- pred } \text{GoalPattern } [ : \text{ Pre } ] [ => \text{ Post } ] [ + \text{ Comp } ].
  \]
  
  (Fields in [...] are optional.)

  - Several form a conjunction (if several match $\rightarrow$ then GLB).
  - Assumed to be *closed* (cover all uses of the predicate).

- Some examples:

  - \text{:- pred qsort(X,Y) => sorted(Y).}
  - \text{:- pred qsort(X,Y) : (intlist(X),var(Y)) => sorted(Y) + no\_fail.}
  - \text{:- pred foo(X,Y) : (ground(X),var(Y)) => (ground(Y),X>Y) + det.}
    \text{:- pred foo(X,Y) : (var(X),ground(Y)) => (ground(X),X>Y).}
Other Assertions

- "Program point" assertions:
  - Properties of program points between literals in clauses.
    
    \[
    \ldots, \text{Literal}, \text{check(Cond)}, \text{Literal}, \ldots
    \]
  - Example:
    
    \[
    p(X) :- q(X,Y), \text{check}((X>Y,Y>=0), r(Y)).
    \]
  - Two types of properties:
    * State properties: refer to current execution state.
    * Forward properties: may refer to future execution states.

    \[
    p(X) :- q(X,Y), \text{check(fwd}((X>0),\text{format}("\sim w \ not \ >0!", [X])), \ldots
    \]

- Many other possibilities.
  E.g. (CIAO) additional optional field for automatic documentation (pl2texi):

    \[
    :- \text{pred GoalPattern} : C \Rightarrow S + G ; \text{Comment}.
    \]
Approximating the Actual Semantics

- Computing the actual program semantics ($[P]$) is generally difficult (the computation process and/or the semantic object are often infinite).
- Program analysis techniques aim at computing approximations of $[P]$.
- One of the most successful, well founded techniques is abstract interpretation [CC77].
- Abstract interpretation techniques have been used for:
  - debugging of imperative languages [Bourdoncle93],
  - diagnosis by proof of logic programs [CLMV96],
  - validation of logic and constraint programs (generation and checking of abstract assertions) [ESOP’96]
- We describe the use of abstract interpretation in abstract validation for arbitrary fixpoint semantics (see [AADEBUG’97] for other applications).
Abstract Interpretation [CC77]

- Compute over abstract values: finite representation of a, possibly infinite, set of values in the concrete domain \(D\).
- An \textit{abstract domain} \(D_{\alpha}\) is the set of all possible abstract semantic values.
- \(D\) and \(D_{\alpha}\) are related via a pair of monotonic mappings \(\langle \alpha, \gamma \rangle\):
  
  - \textit{abstraction} \(\alpha : D \rightarrow D_{\alpha}\)
  - \textit{concretization} \(\gamma : D_{\alpha} \rightarrow D\)

such that:

  - \(\forall x \in D : \gamma(\alpha(x)) \supseteq x\), and
  - \(\forall y \in D_{\alpha} : \alpha(\gamma(y)) = y\).

i.e., \(\langle \alpha, \gamma \rangle\) conform a \textit{Galois insertion} over \(D\) and \(D_{\alpha}\).

- \textit{Example}: (types)
  
  \(D = \{1, 2, 3, \ldots\}, \quad \alpha(\{2, 4\}) = \text{even}, \quad \gamma(\text{even}) = \{2, 4, \ldots\}, \quad \text{even} \ast_{\alpha} \text{even} = \text{even}\)
Abstract Interpretation (cont.)

- An abstract semantic operator $S_P^\alpha$ can be defined which is correct w.r.t. $S_P$.
- $lfp(S_P^\alpha)$ is denoted by $[P]_\alpha$.
- The following relations hold:
  - $\forall x \in D : \gamma(S_P^\alpha(\alpha(x))) \supseteq S_P(x)$
  - $\gamma([P]_\alpha) \supseteq [P]$ equivalently $[P]_\alpha \supseteq \alpha([P])$.
- An abstract operator $S_P^\beta$ is said to be precise if it satisfies:
  - $\gamma([P]_\alpha) = [P]$ equivalently $[P]_\alpha = \alpha([P])$.
- Galois insertions normally over-approximate $[P]$.
- Example: type inference.
- It is possible to work dually (and under-approximate $[P]$) by simply replacing $\supseteq$ with $\subseteq$. 
Assertions in Program Analysis: *Analyzer Output*

- Additional prefix/concepts: `check, true` (and, later `trust`).
- All previous assertions are “check” (i.e., this is default).
- “True” assertions: have been proved to hold. (e.g., output from the analyzer / assertion checker).
  - Example:
    ```prolog
    :- true success p(X) : ground(X).
    ```
  - Also, program point output. Example:
    ```prolog
    p(X,Y):-
     true(ground(X)),
     q(X,Z),
     true((ground(X),ground(Z))),
     r(Z,Y),
     true((ground(X),ground(Y),ground(Z))).
    ```
Guiding the Analysis

- “Entry” assertions: describes external calls to a predicate.
  - Example:
    ```prolog
    :- entry q(X,Y) : (ground(X),var(Y)).
    ```

- “Trust” assertions: have to be assumed to hold. (e.g., guiding the analyzer / assertion checker).
  - Example:
    ```prolog
    :- 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]
Using Analysis in Debugging

- Direct observation of compiler output can detect many bugs:
  - type errors,
  - mode errors (e.g., builtins),
  - infinite loops,
  - possible failure,
  - ...
  - even inefficiencies (cost analysis).

- Objective: automate the process.
Integrated Validation and Diagnosis Based on Approximations

Program

Analyzer (abstract interpreter)

Comparator

RT-test Annotator

Program + RT-tests

:- trust :- entry

Specific. :- check

:- holds

:- check

:- true

:- false

Errors

Declarative / Abstract Diagnoser (or user)
Validation Using Abstract Interpretation [AADEBUG’97]

- Specification given as a semantic value $\mathcal{I}_\alpha \in D_\alpha$ and compared with $[P]_\alpha$.

<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. $\mathcal{I}_\alpha$</td>
<td>$\alpha([P]) \subseteq \mathcal{I}_\alpha$</td>
<td>$[P]<em>{\alpha^+} \subseteq \mathcal{I}</em>\alpha$</td>
</tr>
<tr>
<td>P is complete w.r.t. $\mathcal{I}_\alpha$</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. $\mathcal{I}_\alpha$</td>
<td>$\alpha([P]) \not\subseteq \mathcal{I}_\alpha$</td>
<td>$[P]<em>{\alpha^-} \not\subseteq \mathcal{I}</em>\alpha$, or $[P]<em>{\alpha^+} \cap \mathcal{I}</em>\alpha = \emptyset \land [P]_\alpha \neq \emptyset$</td>
</tr>
<tr>
<td>P is incomplete w.r.t. $\mathcal{I}_\alpha$</td>
<td>$\mathcal{I}_\alpha \not\subseteq \alpha([P])$</td>
<td>$\mathcal{I}<em>\alpha \not\subseteq [P]</em>{\alpha^+}$</td>
</tr>
</tbody>
</table>

$([P]_{\alpha^+}$ represents that $[P]_\alpha \supseteq \alpha([P])$ and $[P]_{\alpha^-}$ indicates that $[P]_\alpha \subseteq \alpha([P])$)

- Conclusions w.r.t. Galois insertions:
  - Suited for proving partial correctness and incompleteness w.r.t. $\mathcal{I}$.
  - It is also possible to prove incorrectness.
  - Completeness can only be proved if the abstraction is “precise.”

- Conclusion w.r.t. reversed Galois insertions:
  - Suited for proving completeness and incorrectness.
  - Partial correctness and incompleteness only if the abstraction is “precise.”
Program construction: an iterative process.

Not only program but also requirements updated \textit{incrementally}. 
Conclusions

- Global analysis w/approximations: important role also in program development.
- It allows going beyond the straight-jacket of traditional types:

<table>
<thead>
<tr>
<th>Types</th>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compulsory (do not allow “any”)</td>
<td>Optional (allow “any”)</td>
</tr>
<tr>
<td>“check”</td>
<td>“check” or “trust”</td>
</tr>
<tr>
<td>Expressed in a Special Language</td>
<td>Expressed in the Source Language</td>
</tr>
<tr>
<td>Limited Property Language</td>
<td>Much More General Property Language</td>
</tr>
<tr>
<td>Limit Programming Language</td>
<td>Do not Limit Programming Language</td>
</tr>
<tr>
<td>Untypable Programs Rejected</td>
<td>Run-time Checks Introduced</td>
</tr>
<tr>
<td>(Almost) Decidable</td>
<td>Approximated</td>
</tr>
</tbody>
</table>

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

- Key issues:

<table>
<thead>
<tr>
<th>Approximation</th>
<th>Suitable assertion language</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract Interpretation</td>
<td>Relating approximations of actual and intended semantics</td>
</tr>
</tbody>
</table>

- See pointers in “http://www.clip.dia.fi.upm.es/” (also, paper(s) in env. WS).