Programming with Global Analysis

M. Hermenegildo[1]

in collaboration with

G. Puebla[1] F. Bueno[1] P. Deransart[2]
W. Drabent[3] G. Ferrand[4] J. Małuszyński[5]

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

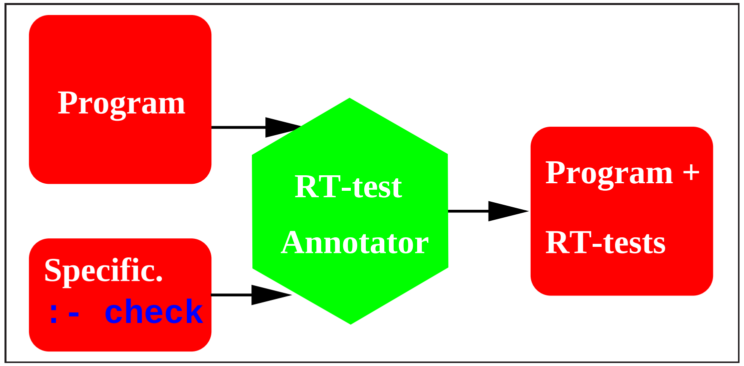
(Partially Supported by ESPRIT Project DiSCiPl:
Debugging Systems for Constraint Programming.)

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).
- Requirements may be:
  - ◇ proved to hold (i.e., they are *verified*),
  - ◇ 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. $\mathcal{I}$ iff $[\![P]\!] \subseteq \mathcal{I}$.
  - P is *complete* w.r.t. $\mathcal{I}$ iff $\mathcal{I} \subseteq [\![P]\!]$.
  - P is *incorrect* w.r.t. $\mathcal{I}$ iff $[\![P]\!] \not\subseteq \mathcal{I}$.
  - P is *incomplete* w.r.t. $\mathcal{I}$ iff $\mathcal{I} \not\subseteq [\![P]\!]$.
- 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:

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

- A subset of the programmer's intentions:

$$\mathcal{I}_1 = \{sorted([X]) \mid X \ is \ an \ integer\}$$

- A superset of the programmer's intentions:

$$\mathcal{I}_2 = \{sorted(L) \mid L \ is \ a \ list \ of \ integers\}$$

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]):

```
:- property sorted/1.                 |  :- typedef list : [];[_|list].
sorted([]).                           |_________________________________
sorted([_]).                          |  :- type integer/1.
sorted([X,Y|Z]) :- X>Y, sorted([Y|Z]).|  % is built-in
______________________________________|_________________________________
:- type list/1.                       |  :- type peano_int/1.
list([]).                             |  peano_int(0).
list([_|Y]) :- list(Y).               |  peano_int(s(X)) :- peano_int(X).
```

Declarative Assertions: *Superset* and *Subset*

- Written by the user, optional — they describe the intended semantics ($\mathcal{I}$).
- Two kinds:
  - "*Superset*" assertions:

    `:- inmodel Pred => Props.`

    - Describe (a superset of) the results of a predicate.
    - Used for *correctness* debugging (violation is a *symptom*).
    - Example:

    ```
    :- inmodel qsort(A,B) => list(B).
    ```
  - "*Subset*" assertions:

    `:- inmodel Pred <= Props.`

    - Describe (a subset of) the results of a predicate.
    - Used for *completeness* debugging (violation is a *symptom*).
    - Example:

    ```
    :- 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:

  `:- success` $Pred$ `=>` $PostCond$`.`

  - ◇ Describe *post-conditions* of a predicate.
  - ◇ Any declarative superset assertion is also a success assertion (due to correctness of the operational semantics).
  - ◇ Example:

    ```
    :- success qsort(A,B) => list(B).
    ```

- Restricting to a subset of calls:

  `:- success` $Pred$ `:` $PreCond$ `=>` $PostCond$`.`

  - ◇ Example:

    ```
    :- success qsort(A,B) : list(A) => list(B).
    ```

Basic Operational Predicate Assertions: *Calls* and *Comp*

- “*Calls*” assertions:

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

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

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

- “*Comp*” assertions:

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

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

```
:- comp qsort(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:

  `:- pred` *GoalPattern* `[ :` *Pre* `] [ =>` *Post* `] [ +` *Comp* `].`

  (Fields in `[...]` are optional.)
  - Several form a conjunction (if several match → then GLB).
  - Assumed to be *closed* (cover all uses of the predicate).
- Some examples:
  - `:- pred qsort(X,Y) => sorted(Y).`
  - `:- pred qsort(X,Y) : (intlist(X),var(Y)) => sorted(Y) + no_fail.`

  -
    ```
    :- pred foo(X,Y) : (ground(X),var(Y)) => (ground(Y),X>Y) + det.
    :- 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.

    $\ldots,\ \mathit{Literal},\ \texttt{check}(\mathit{Cond}),\ \mathit{Literal},\ \ldots$
  - Example:

    ```
    p(X) :- q(X,Y), 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), check(fwd((X>0),format("~w not >0!",[X]))), ...
    ```

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

  $\texttt{:- pred}\ \mathit{GoalPattern}\ \texttt{:}\ \ C\ \texttt{=>}\ S\ \texttt{+}\ G\ \texttt{;}\ \mathit{Comment}\texttt{.}$

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).

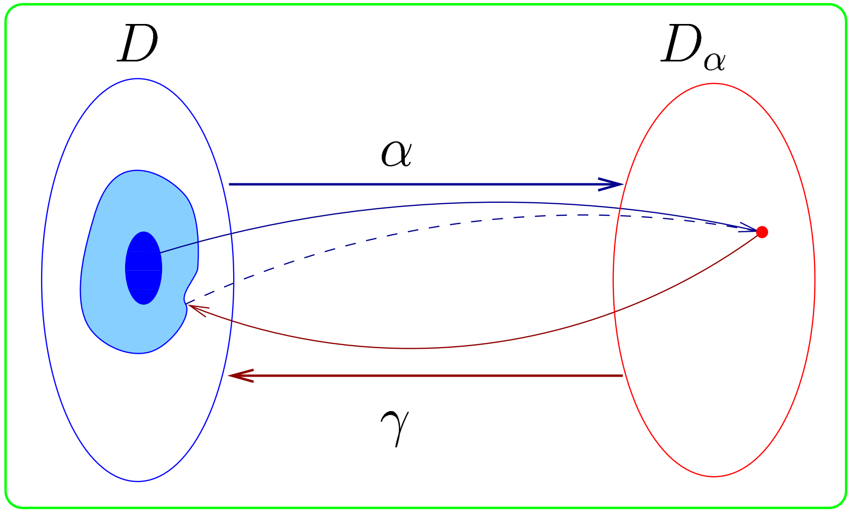
Abstract Interpretation [CC77]

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

  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 *Galois insertion* over D and D_α.



- *Example:* (types)
  $D = \{1, 2, 3, \ldots\}, \;\; \alpha(\{2, 4\}) = even, \;\; \gamma(even) = \{2, 4, \ldots\}, \;\; even *_\alpha even = even$

Abstract Interpretation (cont.)

- An abstract semantic operator S_P^α 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^α 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:

    ```
    :- true success p(X) : ground(X).
    ```

  - Also, program point output. Example:

```
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:

```
:- entry q(X,Y) : (ground(X),var(Y)).
```

- “Trust” assertions: have to be assumed to hold. (e.g., guiding the analyzer / assertion checker).
  - Example:

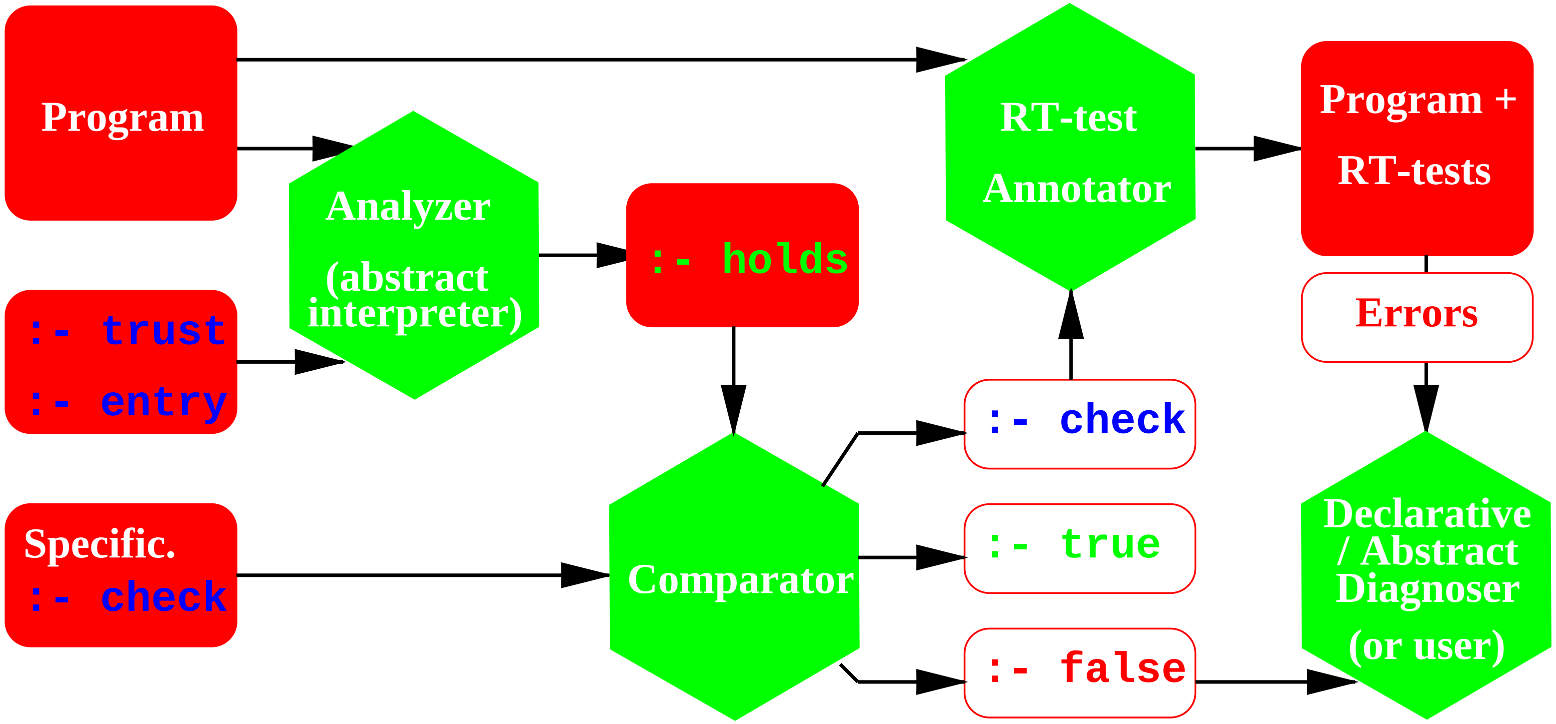
```
:- trust success p(X) : ground(X).
```

  - 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



Validation Using Abstract Interpretation [AADEBUG'97]

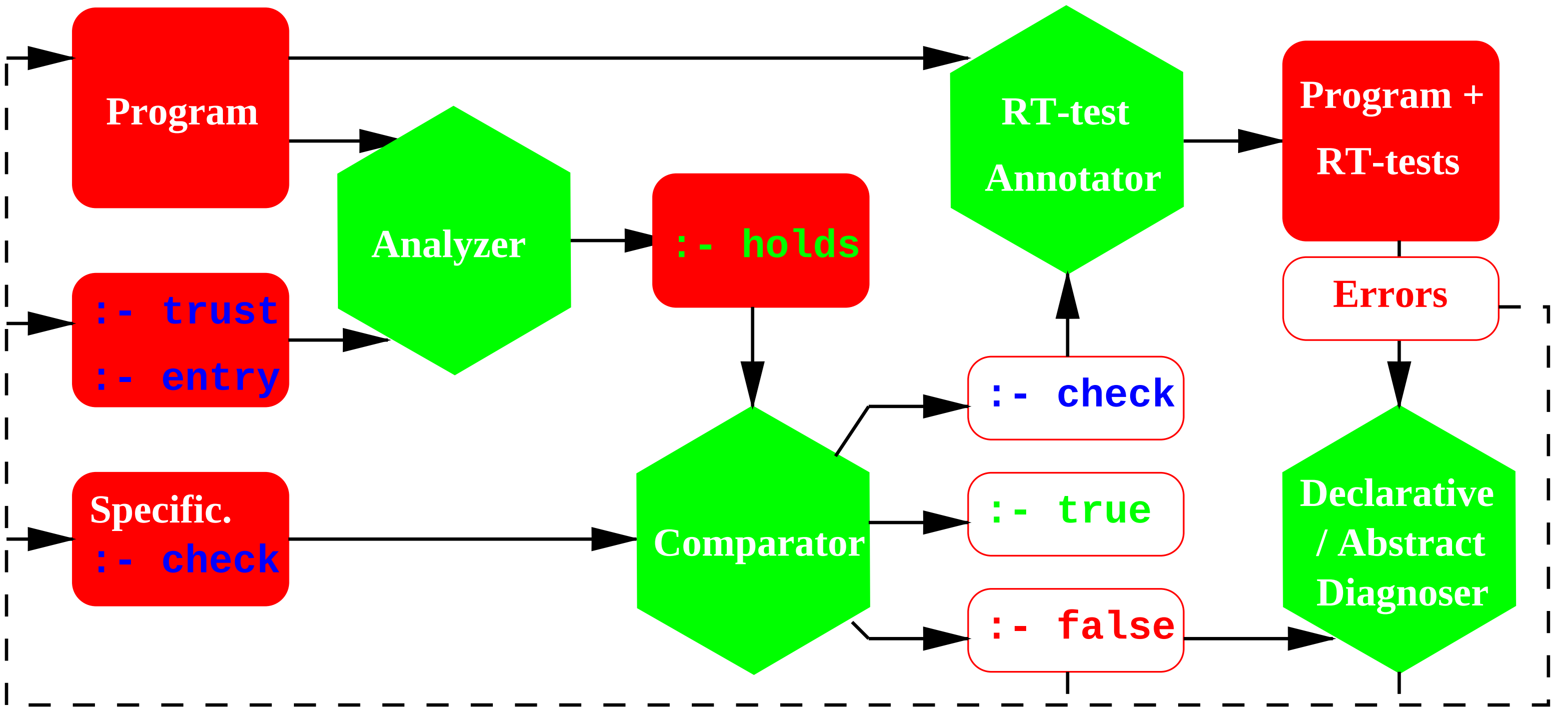
- Specification given as a semantic value $\mathcal{I}_\alpha \in D_\alpha$ and compared with $[\![P]\!]_\alpha$.

Property	**Definition**	**Sufficient condition**
P is partially correct w.r.t. $\mathcal{I}_\alpha$	$\alpha([\![P]\!]) \subseteq \mathcal{I}_\alpha$	$[\![P]\!]_{\alpha^+} \subseteq \mathcal{I}_\alpha$
P is complete w.r.t. $\mathcal{I}_\alpha$	$\mathcal{I}_\alpha \subseteq \alpha([\![P]\!])$	$\mathcal{I}_\alpha \subseteq [\![P]\!]_{\alpha^-}$
P is incorrect w.r.t. $\mathcal{I}_\alpha$	$\alpha([\![P]\!]) \not\subseteq \mathcal{I}_\alpha$	$[\![P]\!]_{\alpha^-} \not\subseteq \mathcal{I}_\alpha$, or $[\![P]\!]_{\alpha^+} \cap \mathcal{I}_\alpha = \emptyset \wedge [\![P]\!]_\alpha \neq \emptyset$
P is incomplete w.r.t. $\mathcal{I}_\alpha$	$\mathcal{I}_\alpha \not\subseteq \alpha([\![P]\!])$	$\mathcal{I}_\alpha \not\subseteq [\![P]\!]_{\alpha^+}$

($[\![P]\!]_{\alpha^+}$ 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."

The feedback loop



- Program construction: an iterative process.
- Not only program but also requirements updated *incrementally*.

Conclusions

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

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

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

- Key issues:

Approximation	Suitable assertion language
Abstract Interpretation	Relating aproximations of actual and intended semantics

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