Static Analysis-based Debugging, Certification, Testing, and Optimization with CiaoPP

Manuel Hermenegildo[1,2]

(with F. Bueno,[2] G. Puebla,[2] M. Carro,[2] P. López-García,[4,1] J. Morales,[3] E. Mera,[3] J. Navas,[5] R. Haemmerlé,[1] M. Méndez,[5] A. Casas,[5] J. Correas,[3] E. Albert,[3] P. Arenas[3])

[1]IMDEA Software Institute, Spain

[2]Technical University of Madrid – UPM, Spain

[3]Complutense University of Madrid – UCM, Spain

[4]Spanish Research Council (CSIC), Spain

[5]CS and EECE Depts., U. of New Mexico, USA

TAPAS 2010 (SAS) – Perpignan, France – Sept. 17, 2010

Objective

- *Facilitate the development of safe, efficient programs*.
- Approach:
  - Next-generation, higher-level, *multiparadigm* prog. languages.
  - *Improved program development environments.*
    - A framework (CiaoPP) which *integrates*:
      - Debugging.
      - Verification and certification.
      - Testing.
      - Optimization (optimized compilation, parallelization, ...).

Objective

- *Facilitate the development of safe, efficient programs*.
- Approach:
  - Next-generation, higher-level, *multiparadigm* prog. languages.
  - *Improved program development environments.*
    - A framework (CiaoPP) which *integrates*:
      - Debugging.
      - Verification and certification.
      - Testing.
      - Optimization (optimized compilation, parallelization, ...).

Verification, Diagnosis, and Safe Approximations

- Need to compare actual semantics $[\![P]\!]$ with intended semantics $\mathcal{I}$:

P is *partially correct* w.r.t. $\mathcal{I}$ iff	$[\![P]\!] \leq \mathcal{I}$
P is *complete* w.r.t. $\mathcal{I}$ iff	$\mathcal{I} \leq [\![P]\!]$
P is *incorrect* w.r.t. $\mathcal{I}$ iff	$[\![P]\!] \not\leq \mathcal{I}$
P is *incomplete* w.r.t. $\mathcal{I}$ iff	$\mathcal{I} \not\leq [\![P]\!]$

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

- *Problem:* difficulty computing $[\![P]\!]$ w.r.t. interesting observables.
- *Approach:* use a *safe approximation* of $[\![P]\!] \rightarrow$ i.e., $[\![P]\!]_{\alpha^+}$ or $[\![P]\!]_{\alpha^-}$
- Specially attractive if compiler computes (most of) $[\![P]\!]_{\alpha^+}$ anyway.

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

[BDD+97, HPB99, PBH00c, PBH00a, HPBLG03]

Verification, Diagnosis, and Safe Approximations

- Need to compare actual semantics $[\![P]\!]$ with intended semantics $\mathcal{I}$:

P is *partially correct* w.r.t. $\mathcal{I}$ iff	$[\![P]\!] \leq \mathcal{I}$
P is *complete* w.r.t. $\mathcal{I}$ iff	$\mathcal{I} \leq [\![P]\!]$
P is *incorrect* w.r.t. $\mathcal{I}$ iff	$[\![P]\!] \not\leq \mathcal{I}$
P is *incomplete* w.r.t. $\mathcal{I}$ iff	$\mathcal{I} \not\leq [\![P]\!]$

  Usually, partial descriptions of $\mathcal{I}$ available, typically as *assertions*.
- *Problem:* difficulty computing $[\![P]\!]$ w.r.t. interesting observables.
- *Approach:* use a *safe approximation* of $[\![P]\!] \rightarrow$ i.e., $[\![P]\!]_{\alpha^+}$ or $[\![P]\!]_{\alpha^-}$
- Specially attractive if compiler computes (most of) $[\![P]\!]_{\alpha^+}$ anyway.

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

[BDD+97, HPB99, PBH00c, PBH00a, HPBLG03]

Verification, Diagnosis, and Safe Approximations

- Need to compare actual semantics $[\![P]\!]$ with intended semantics $\mathcal{I}$:

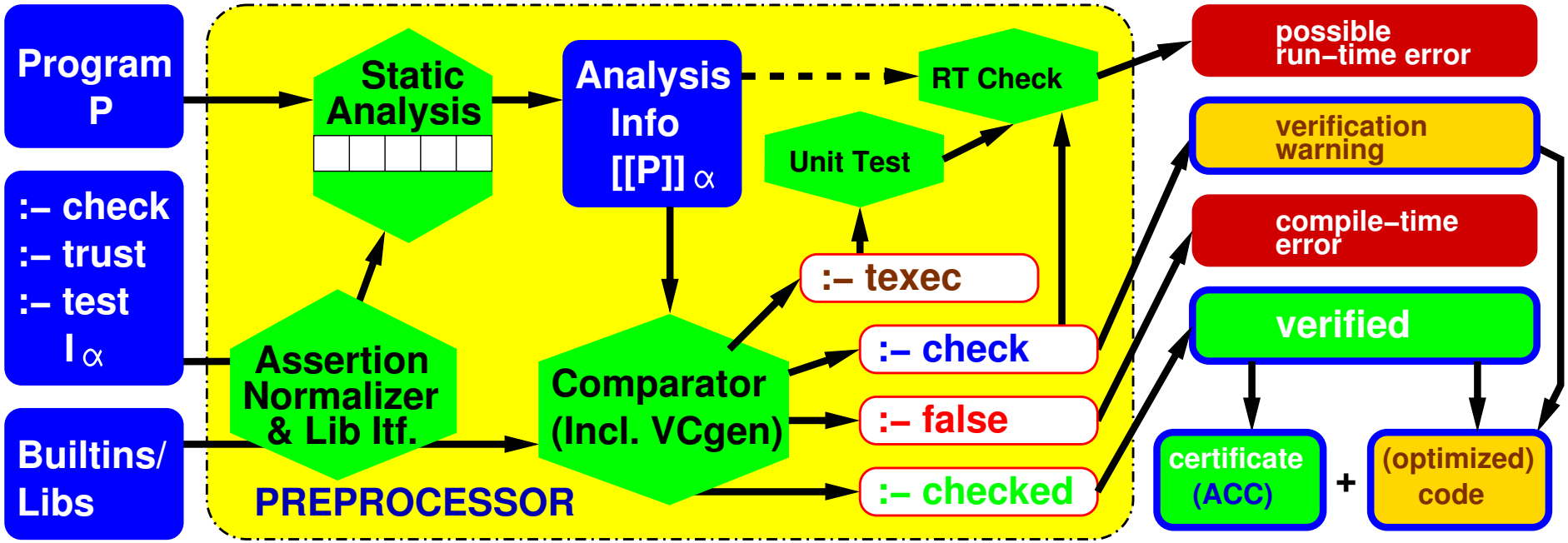
P is *partially correct* w.r.t. $\mathcal{I}$ iff	$[\![P]\!] \leq \mathcal{I}$
P is *complete* w.r.t. $\mathcal{I}$ iff	$\mathcal{I} \leq [\![P]\!]$
P is *incorrect* w.r.t. $\mathcal{I}$ iff	$[\![P]\!] \not\leq \mathcal{I}$
P is *incomplete* w.r.t. $\mathcal{I}$ iff	$\mathcal{I} \not\leq [\![P]\!]$

  Usually, partial descriptions of $\mathcal{I}$ available, typically as *assertions*.
- *Problem:* difficulty computing $[\![P]\!]$ w.r.t. interesting observables.
- *Approach:* use a *safe approximation* of $[\![P]\!] \rightarrow$ i.e., $[\![P]\!]_{\alpha^+}$ or $[\![P]\!]_{\alpha^-}$
- Specially attractive if compiler computes (most of) $[\![P]\!]_{\alpha^+}$ anyway.

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

[BDD+97, HPB99, PBH00c, PBH00a, HPBLG03]

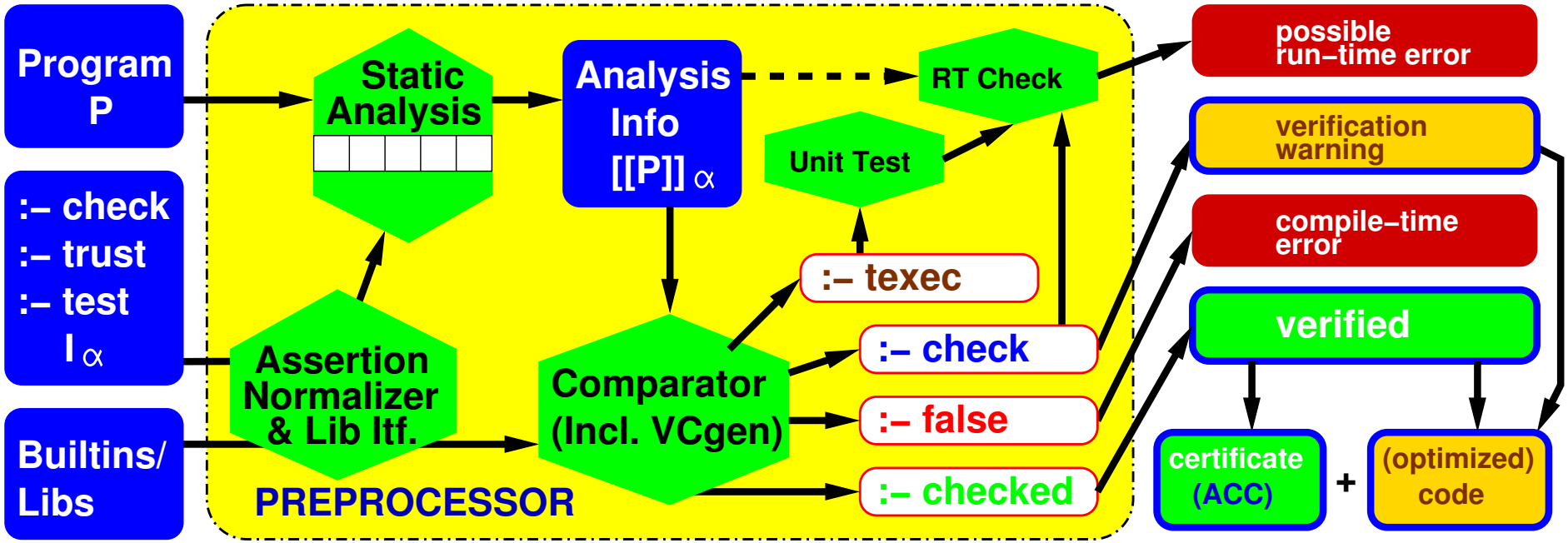
The CiaoPP Framework



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

[BDD+97, HPB99, PBH00c, PBH00a, HPBLG03, APH05, MLGH09]

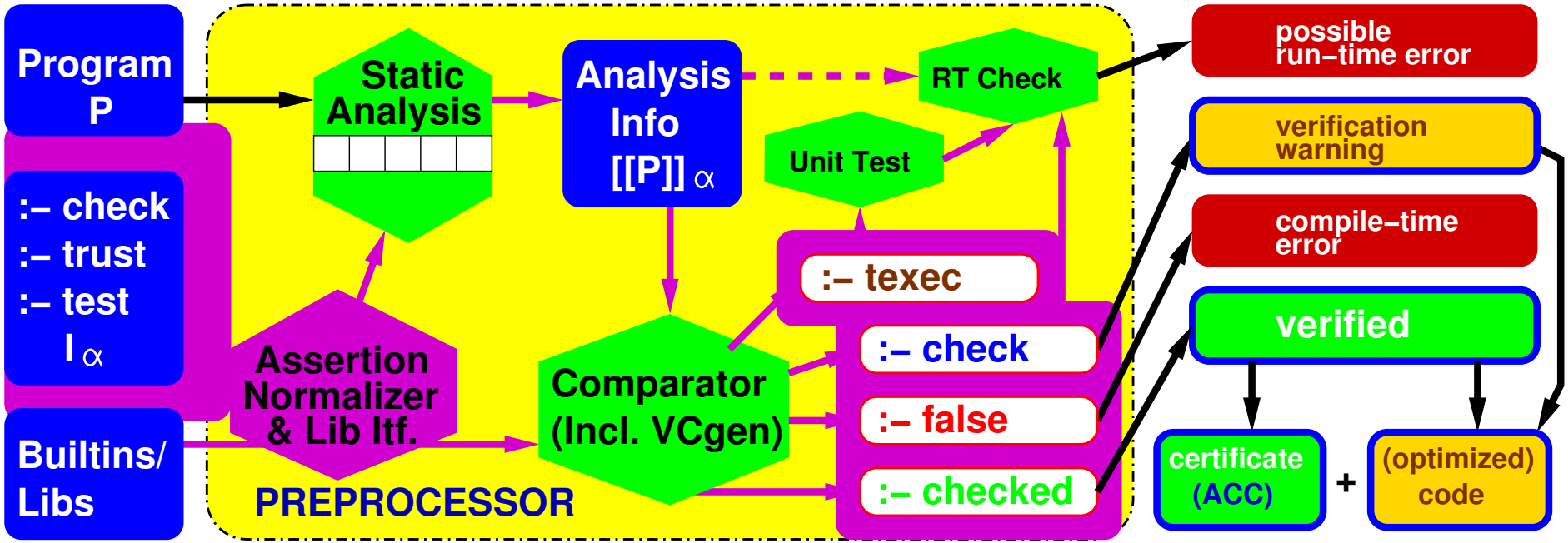
The CiaoPP Framework



- Java source / bytecode.
- Ciao (multi-paradigm):
  - Functions (including higher-order, types, etc.).
  - Predicates (unification, search, including ISO-Prolog).
  - Constraints.
  - Objects and imperative constructs.

[BDD+97, HPB99, PBH00c, PBH00a, HPBLG03, APH05, MLGH09]

The Assertion Language



- Assertions optional, can be added at any time.
- *Sets* of pre/post/global triples (+ "status" field, documentation, ...).
- Use everywhere, for many purposes (including doc generation).
- Make it worthwhile to the programmer to include them.
- Part of the programming language and "runnable" (Ciao).

[BDD+97, PBH97, HPB99, PBH00b, MLGH09]

The Assertion Language (*simplified* grammar, Java)

$$
\begin{array}{lcl}
\langle primitive\_assrt \rangle & ::= & primitive\_name(var^*)\langle assrt \rangle^* \\
\langle assrt \rangle & ::= & \texttt{@requires (} \langle prop \rangle^* \texttt{)} \\
 & | & \texttt{@ensures (} \langle prop \rangle^* \texttt{)} \\
 & | & \texttt{@cost (} \langle resource\_usage \rangle^* \texttt{)} \\
 & | & \texttt{@if (} \langle prop \rangle^* \texttt{) \{ } \langle prop \rangle^* \texttt{ \} [cost (} \langle resource\_usage \rangle^* \texttt{)]} \\
\langle resource\_usage \rangle & ::= & \text{res\_usage}(res\_name, \langle expr \rangle) \\
 & & \\
\langle prop \rangle & ::= & type \\
 & | & \text{size}(var, \langle sz\_metric \rangle, \langle expr \rangle) \\
 & | & \text{size\_metric}(var, \langle sz\_metric \rangle) \\
 & & \\
\langle expr \rangle & ::= & \langle expr \rangle \langle bin\_op \rangle \langle expr \rangle \mid (\sum \mid \prod) \langle expr \rangle \\
 & | & \langle expr \rangle^{\langle expr \rangle} \mid \log_{num} \langle expr \rangle \mid -\langle expr \rangle \\
 & | & \langle expr \rangle ! \mid \infty \mid \text{num} \\
 & | & \text{size}([\langle sz\_metric \rangle,]\text{arg(r num)}) \\
\langle bin\_op \rangle & ::= & + \mid - \mid \times \mid / \mid \% \\
 & & \\
\langle sz\_metric \rangle & ::= & \text{int} \mid \text{ref} \mid \ldots
\end{array}
$$

The Assertion Language (Grammar, Ciao)

$$
\begin{array}{lcl}
\langle program\_assrt \rangle & ::= & \textbf{:-}\ \langle status\_flag \rangle\ \langle pred\_assrt \rangle . \\
 & | & \textbf{:-}\ \textbf{head\_cost}(\langle approx \rangle, \text{Res\_name}, \Delta^{H}). \\
 & | & \textbf{:-}\ \textbf{literal\_cost}(\langle approx \rangle, \text{Res\_name}, \Delta^{L}). \\
\langle status\_flag \rangle & ::= & \textbf{trust} \mid \textbf{check} \mid \textbf{true} \mid \epsilon \\
\langle pred\_assrt \rangle & ::= & \textbf{pred}\ \langle pred\_desc \rangle\ \langle pre\_cond \rangle\ \langle post\_cond \rangle\ \langle comp\_cond \rangle . \\
\langle pred\_desc \rangle & ::= & \text{Pred\_name} \mid \text{Pred\_name}(\langle args \rangle) \\
\langle args \rangle & ::= & \text{Var} \mid \text{Var}, \langle args \rangle \\
\langle pre\_cond \rangle & ::= & : \langle state\_props \rangle \mid \epsilon \\
\langle post\_cond \rangle & ::= & => \langle state\_props \rangle \mid \epsilon \\
\langle comp\_cond \rangle & ::= & + \langle comp\_props \rangle \mid \epsilon \\
\langle state\_prop \rangle & ::= & \textbf{size}(\text{Var}, \langle approx \rangle, \langle sz\_metric \rangle, \langle arith\_expr \rangle) \mid \text{State\_prop} \\
\langle state\_props \rangle & ::= & \langle state\_prop \rangle \mid \langle state\_prop \rangle, \langle state\_props \rangle \\
\langle comp\_prop \rangle & ::= & \textbf{size\_metric}(\text{Var}, \langle sz\_metric \rangle) \mid \langle cost \rangle \mid \text{Comp\_prop} \\
\langle comp\_props \rangle & ::= & \langle comp\_prop \rangle \mid \langle comp\_prop \rangle, \langle comp\_props \rangle \\
\langle cost \rangle & ::= & \textbf{cost}(\langle approx \rangle, \text{Res\_name}, \langle arith\_expr \rangle) \\
\langle approx \rangle & ::= & \textbf{ub} \mid \textbf{lb} \mid \textbf{oub} \mid \textbf{olb} \\
\langle sz\_metric \rangle & ::= & \textbf{value} \mid \textbf{length} \mid \textbf{size} \mid \textbf{void} \\
\langle arith\_expr \rangle & ::= & - \langle arith\_expr \rangle \mid \langle arith\_expr \rangle\ ! \mid \langle quantifier \rangle\ \langle arith\_expr \rangle \\
 & | & \langle arith\_expr \rangle\ \langle bin\_op \rangle\ \langle arith\_expr \rangle \\
 & | & \langle arith\_expr \rangle^{\langle arith\_expr \rangle} \mid log_{Num}\ \langle arith\_expr \rangle \\
 & | & \text{Num} \mid \langle sz\_metric \rangle(\text{Var}) \\
\langle bin\_op \rangle & ::= & + \mid - \mid * \mid / \\
\langle quantifier \rangle & ::= & \sum \mid \prod
\end{array}
$$

The Ciao Assertion Language

```
:- pred Pred [:Precond] [=> Postcond] [+ Comp-formula ] .
```

Each typically a "mode" of use; the set *covers the valid calls*.

```
:- pred qs(X,Y) : list(int) * var => sorted(Y) + (det,not_fails).
:- pred qs(X,Y) : var * list(int) => ground(X) + not_fails.
```

Properties (from libraries or user defined):

```
:- regtype color := green | blue | red.
:- regtype list(X) := [] | [X|list].
:- prop sorted := [] | [ _ ] | [X,Y|Z] :- X > Y, sorted([Y|Z]).
```

Program-point Assertions

- Property calls inlined with code: `..., check( X>0 ), ...`.

Assertion Status

- Each assertion has prefix **check**, **trust**, **true**, **false**, etc. –its "status."

The Ciao Assertion Language

```
:- pred Pred [:Precond] [=> Postcond] [+ Comp-formula ] .
```

Each typically a "mode" of use; the set *covers the valid calls*.

```
:- pred qs(X,Y) : list(int) * var => sorted(Y) + (det,not_fails).
:- pred qs(X,Y) : var * list(int) => ground(X) + not_fails.
```

Properties (from libraries or user defined):

```
:- regtype color := green | blue | red.
:- regtype list(X) := [] | [X|list].
:- prop sorted := [] | [ _ ] | [X,Y|Z] :- X > Y, sorted([Y|Z]).
```

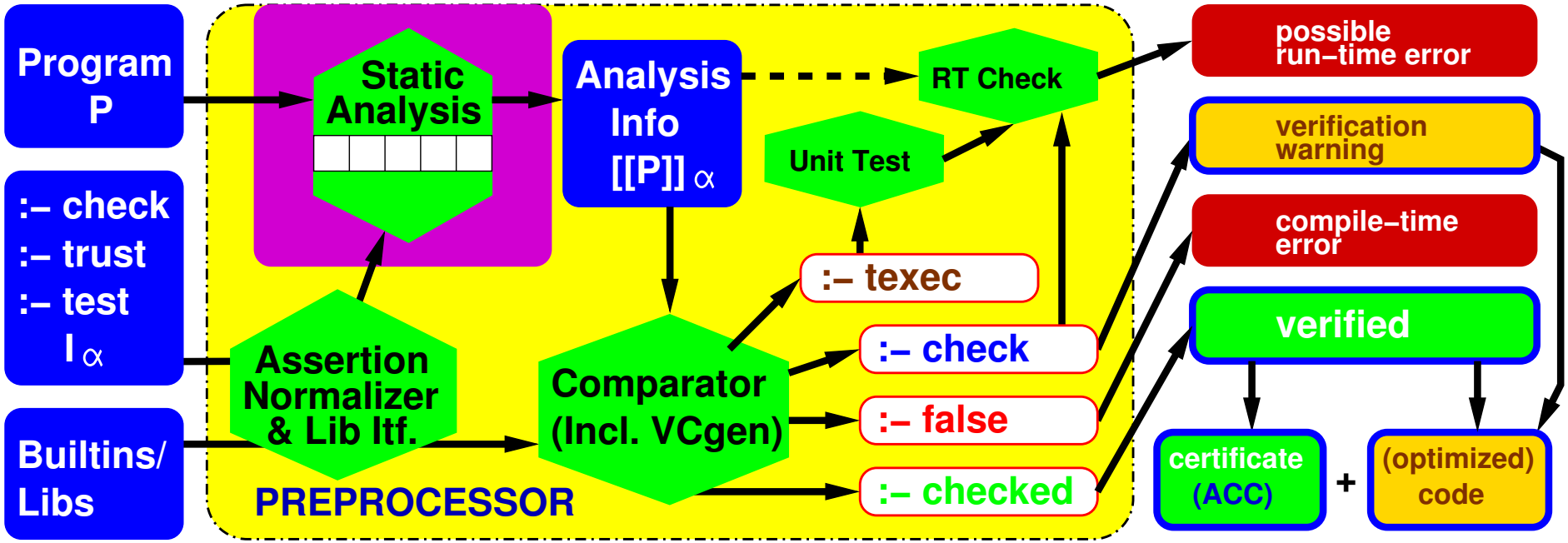
Program-point Assertions

- Property calls inlined with code: `..., check( X>0 ), ....`

Assertion Status

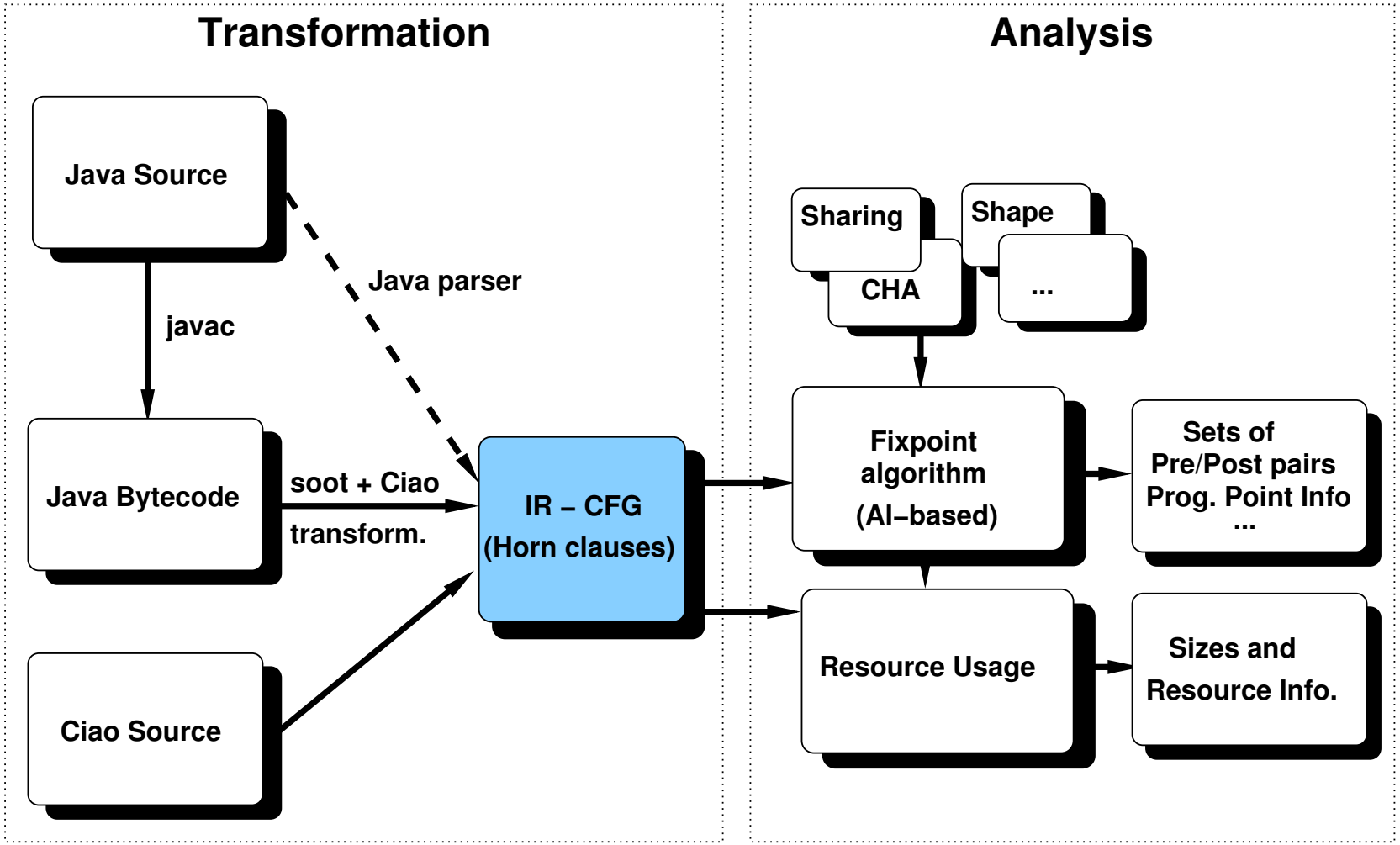
- Each assertion has prefix **check**, **trust**, **true**, **false**, etc. –its "status."

The Analyses



- Modular, parametric, polyvariant abstract interpretation.
- Accelerated, incremental fixpoint.
- Properties:
  - Shapes, data sizes, sharing/aliasing, CHA, determinacy, exceptions, termination, ...
  - Resources (time, memory, energy, ...), (user-defined) resources.

Starting Point: The Intermediate Representation



[MLNH07]

Intermediate Representation

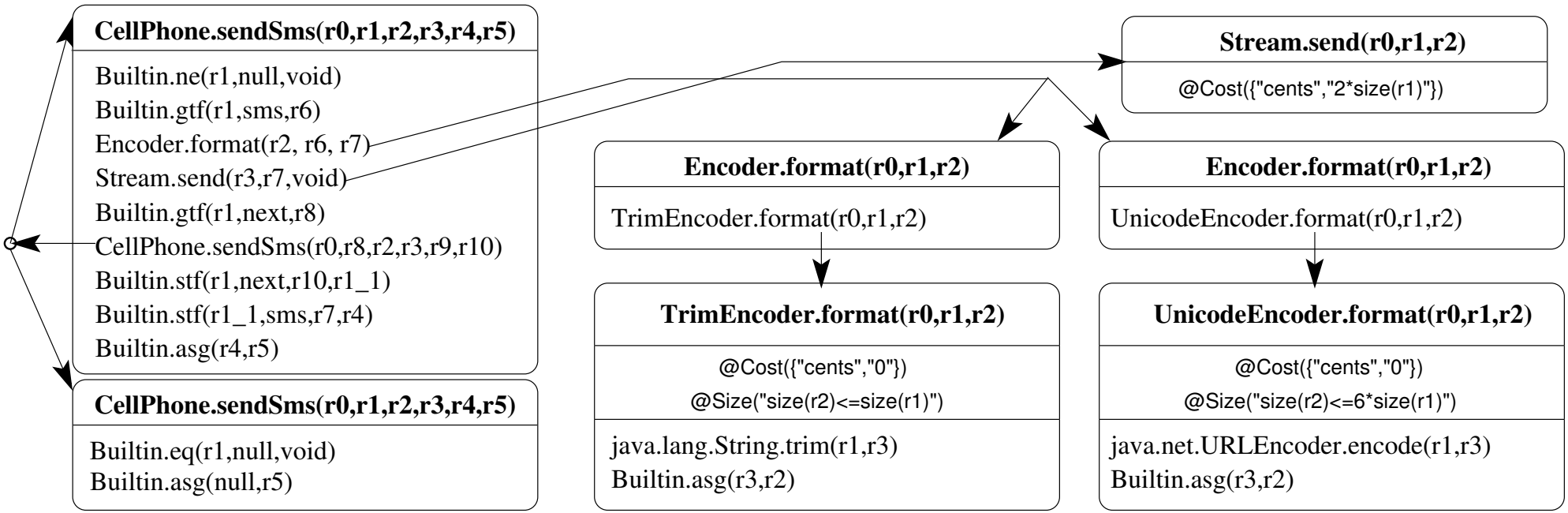
- Used for all analyses: aliasing, CHA, resources, shape/types, data sizes, etc.
- Used to support several languages / paradigms.
- Based on "blocks" (each block represented as a *Horn clause*).
- E.g., for Java:
  - Elimination of stack variables.
  - Conversion to three-address statements.
  - SSA transformation (e.g., splitting of input/output param).
  - Explicit representation of this and ret as extra block parameters.
  - Conversion of loops into recursions among blocks.
  - Branching, cases, and dynamic dispatch $\rightarrow$ blocks w/same signature.
  - Generation of block-based CFG.
  - Conversion to horn clauses for easier manipulation.

Example: sending SMSs

```
public class CellPhone {

 void sendSms(SmsPacket smsPk,
              Encoder enc,
              Stream stm) {
 if (smsPk != null) {
  stm.send(
        enc.format(smsPk.sms));
  sendSms(smsPk.next,enc,stm);
 }}}

class SmsPacket{
   String sms;
   SmsPacket next;
}

abstract class Stream{
   @Cost({"cents","2*size(data)"})
   native void send(String data);
}
```

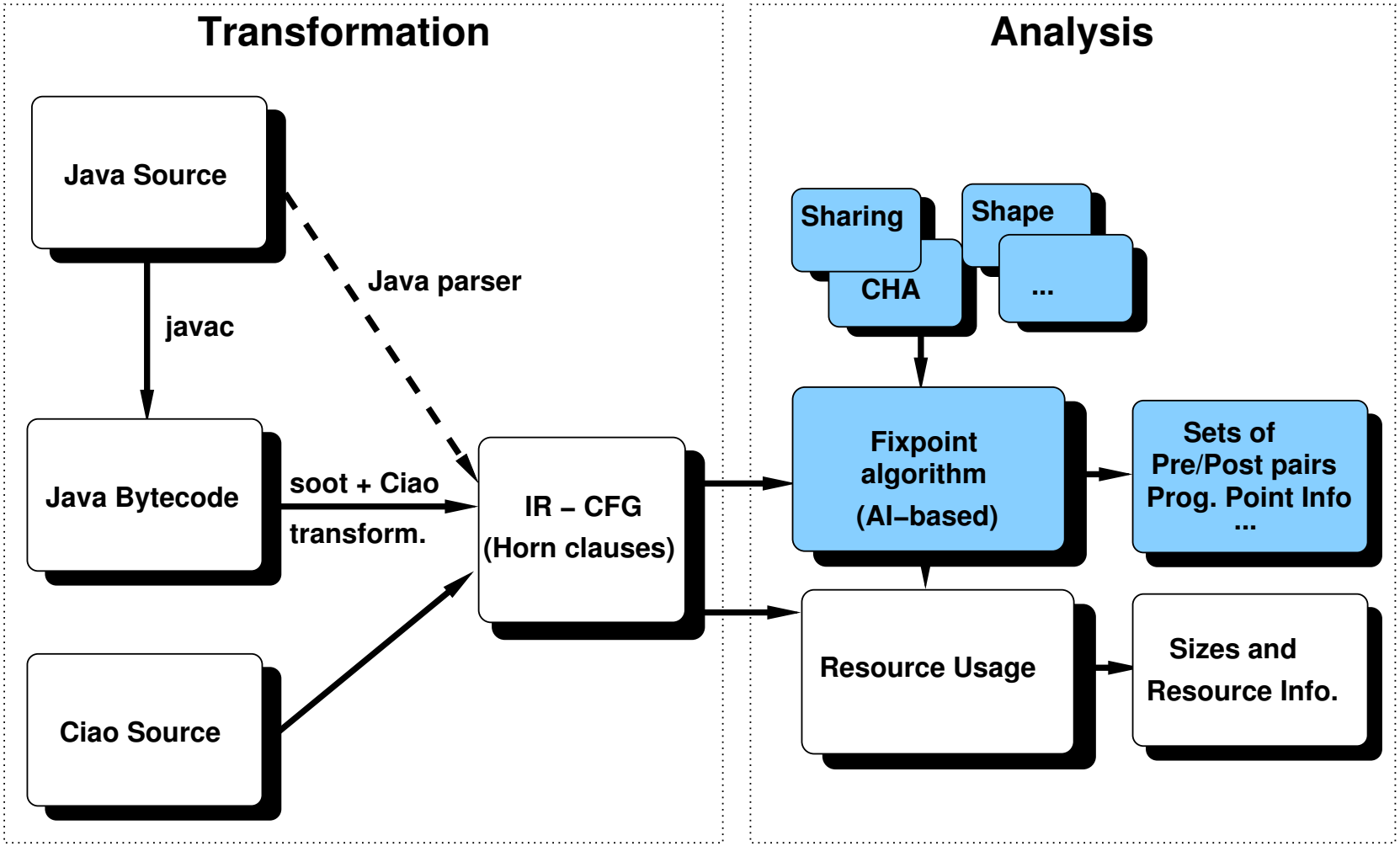
```
interface Encoder{
   String format(String data);
   }

class TrimEncoder implements Encoder{
   @Cost({"cents","0"})
   @Size("size(ret)<=size(s)")
   public String format(String s){
     return s.trim();
   }}

class UnicodeEncoder implements Encoder{
   @Cost({"cents","0"})
   @Size("size(ret)<=6*size(s)")
   public String format(String s){
     return java.net.URLEncoder.encode(s)
   }}
```

Example: sending SMSs – IR



- Internal representation: basic block → Horn clause.
- Annotations (since Java 1.5) are preserved in the bytecode so they can be carried over to our IR.

Fixpoint-based Analyzers



[MH92, BGH99, PH96, HPMS00, NMLH07]

[MGH94, BCHP96, PH00, BdlBH+01, PCPH06, PCPH08]

An Efficient, Parametric Fixpoint Algorithm

- Computes $\mathrm{lfp}(S_P^\alpha) = [\![P]\!]_\alpha$, s.t. $[\![P]\!]_\alpha$ safely approximates $[\![P]\!]$.
- It maintains and computes as a result (simplified):
  - **An answer table**: $\{block : \lambda_{in} \mapsto \lambda_{out}\}$.
    - Exit states for calls to *block* satisfying precond λ_{in} meet postcond λ_{out}.
  - **A dependency arc table**: $\{A : \lambda_{inA} \Rightarrow B : \lambda_{inB}\}$.
    - Answers for call $A : \lambda_{inA}$ depend on the answers for $B : \lambda_{inB}$: (if exit for $B : \lambda_{inB}$ changes, exit for $A : \lambda_{inA}$ possibly also changes).
    - $Dep(B : \lambda_{inB})$ = the set of entries depending on $B : \lambda_{inB}$.
- Characteristics:
  - **Precision:** context-sensitivity / multivariance, prog. point info, ...
  - **Efficiency:** memoization, dependency tracking, SCCs, base cases, ...
  - **Genericity:** abstract domains are plugins, configurable, ...
  - Handles mutually recursive methods.
  - Handles library calls (essential for Java), externals, ...
  - Modular, incremental.

Generic framework for implementing analyses / generating certificates.

An Efficient, Parametric Fixpoint Algorithm

- Computes $\mathrm{lfp}(S_P^\alpha) = [\![P]\!]_\alpha$, s.t. $[\![P]\!]_\alpha$ safely approximates $[\![P]\!]$.
- It maintains and computes as a result (simplified):
  - **An answer table**: $\{block : \lambda_{in} \mapsto \lambda_{out}\}$.
    - Exit states for calls to *block* satisfying precond λ_{in} meet postcond λ_{out}.
  - **A dependency arc table**: $\{A : \lambda_{inA} \Rightarrow B : \lambda_{inB}\}$.
    - Answers for call $A : \lambda_{inA}$ depend on the answers for $B : \lambda_{inB}$: (if exit for $B : \lambda_{inB}$ changes, exit for $A : \lambda_{inA}$ possibly also changes).
    - $Dep(B : \lambda_{inB})$ = the set of entries depending on $B : \lambda_{inB}$.
- Characteristics:
  - **Precision:** context-sensitivity / multivariance, prog. point info, ...
  - **Efficiency:** memoization, dependency tracking, SCCs, base cases, ...
  - **Genericity:** abstract domains are plugins, configurable, ...
  - Handles mutually recursive methods.
  - Handles library calls (essential for Java), externals, ...
  - Modular, incremental.

Generic framework for implementing analyses / generating certificates.

An Efficient, Parametric Fixpoint Algorithm

- Computes $\mathrm{lfp}(S_P^\alpha) = [\![P]\!]_\alpha$, s.t. $[\![P]\!]_\alpha$ safely approximates $[\![P]\!]$.
- It maintains and computes as a result (simplified):
  - **An answer table**: $\{block : \lambda_{in} \mapsto \lambda_{out}\}$.
    - Exit states for calls to *block* satisfying precond λ_{in} meet postcond λ_{out}.
  - **A dependency arc table**: $\{A : \lambda_{inA} \Rightarrow B : \lambda_{inB}\}$.
    - Answers for call $A : \lambda_{inA}$ depend on the answers for $B : \lambda_{inB}$: (if exit for $B : \lambda_{inB}$ changes, exit for $A : \lambda_{inA}$ possibly also changes).
    - $Dep(B : \lambda_{inB})$ = the set of entries depending on $B : \lambda_{inB}$.
- Characteristics:
  - **Precision:** context-sensitivity / multivariance, prog. point info, ...
  - **Efficiency:** memoization, dependency tracking, SCCs, base cases, ...
  - **Genericity:** abstract domains are plugins, configurable, ...
  - Handles mutually recursive methods.
  - Handles library calls (essential for Java), externals, ...
  - Modular, incremental.

Generic framework for implementing analyses / generating certificates.

CFG traversal

- Blocks are nodes; edges are invocations.
- Top-down traversal of this CFG, starting from entry point.
- Within each block: sequence of builtins, handled in the domain.
- Inter-block calls/edges: *project*, *extend*, etc. (next slide).
- As graph is traversed, triples $(block, \lambda_{in}, \lambda_{out})$ are stored for each block in a *memo table*.
- Memo table entries have status $\in \{fixpoint, approx., complete\}$.
- Iterate until all *complete*.

Interprocedural analysis / reursion support

- **Project** the caller state over the actual parameters,
- find all the **compatible implementations** (blocks),
- **rename** to their formal parameters,

... abstractly execute each compatible block, ...

- calculate the **least upper bound** of the partial results of each block (if "monovariant on success" flag),
- **rename back** to the actual parameters and, finally
- **extend** (reconcile) return state into calling state.

Speeding up convergence

- Analyze non-recursive blocks first,
  use as starting λ_{out} in recursions.
- Blocks derived from conditionals treated specially
  (no *project* or *extend* operations required).
- The $(block, \lambda_{in}, \lambda_{out})$ tuples
  act as a cache that avoids recomputation.
- Use strongly-connected components (on the fly).

Domain examples

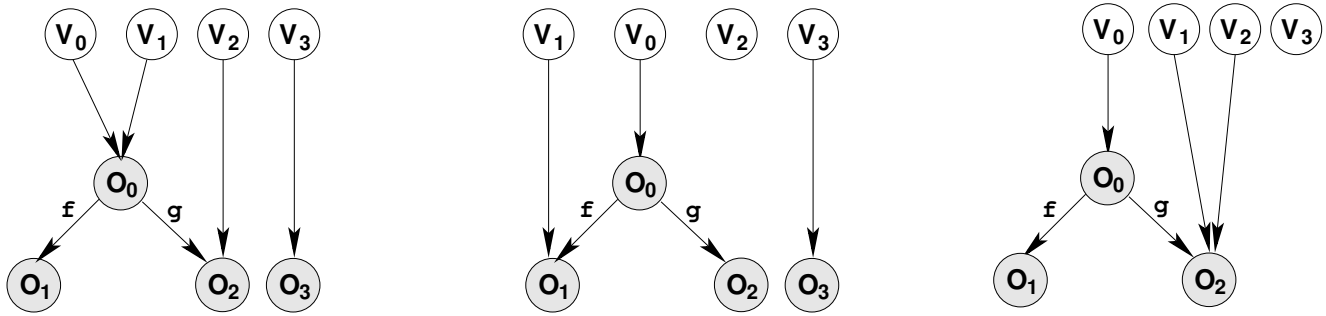
Quite a number of domains: shape, aliasing, nullity, CHA, polyhedra, data sizes, depth-k, determinacy, termination, non-failure, ...

Set-sharing (non-aliasing + nullity):

- Uses set of sets of variables to approximate all possible sharing that occur at a given program point (plus nullity):

$$SH_p = \{\{v_0, v_1\}, \{v_0, v_1, v_2\}, \{v_3\}\}$$

"v_0 may share with v_1 & v_2, or just v_1; v_3 may point to a non-null loc."



Analysis ensures that v_3 definitely does not share with v_0, or v_1, or v_2.

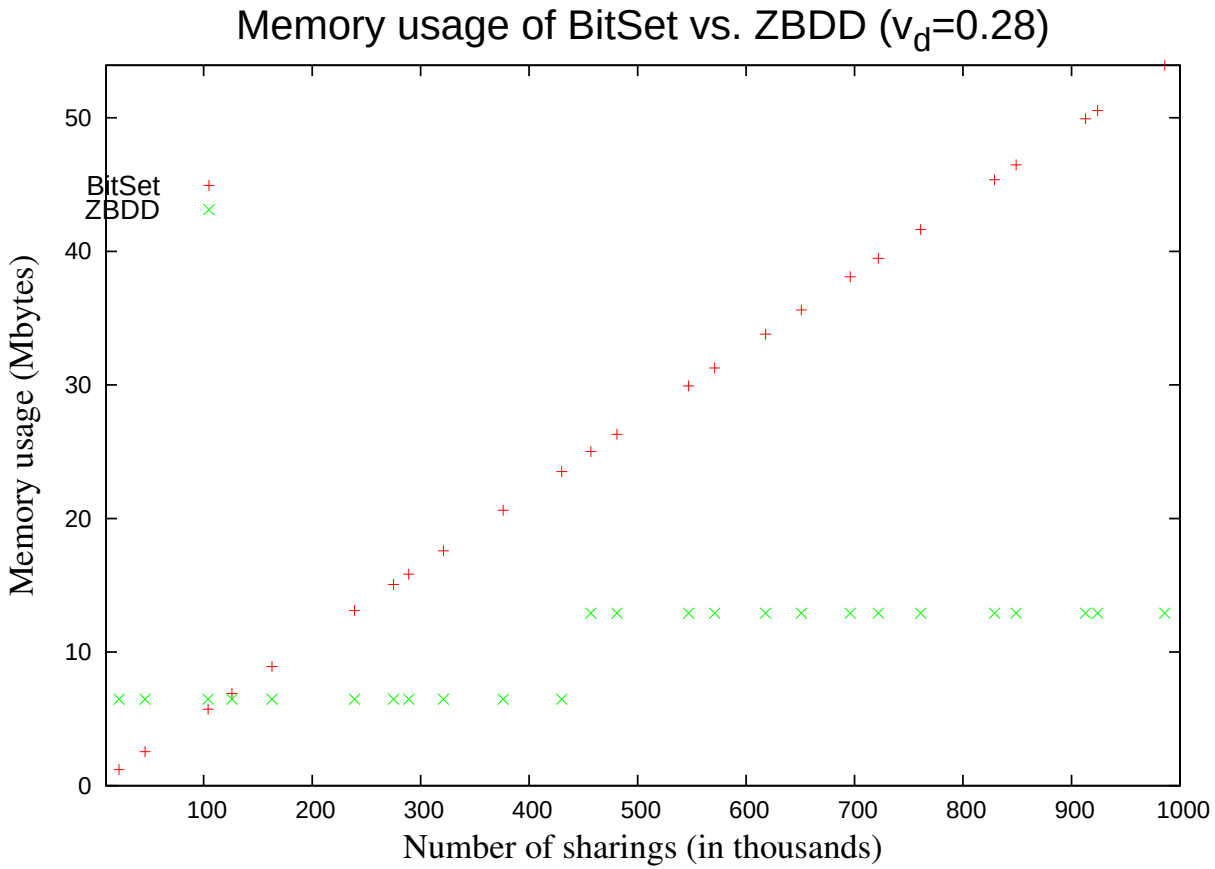
- Much work optimizing it: using ZBDDs, negative representations, etc.

[MH89, MH91, DLGH97, VB02, BLGH04, LGBH05, NBH06, MSHK07]

[MLH08, MKSH08, MMLH+08, MHKS08, MKH09, LGBH10]

Sharing, experimental results, memory usage

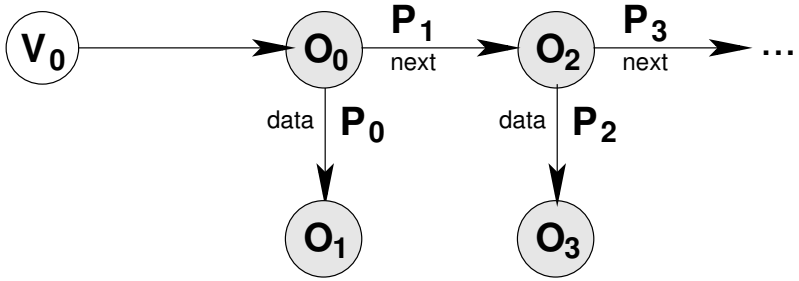
`BitSet` vs. `ZBDD`-based implementation.



[MLLH08]

Shape+set sharing

- Sharing can be combined with *structural* information. Set sharing can talk about *any* pointer (not just local vars). E.g., this linked list:



Can be abstracted as ("depth-k"):

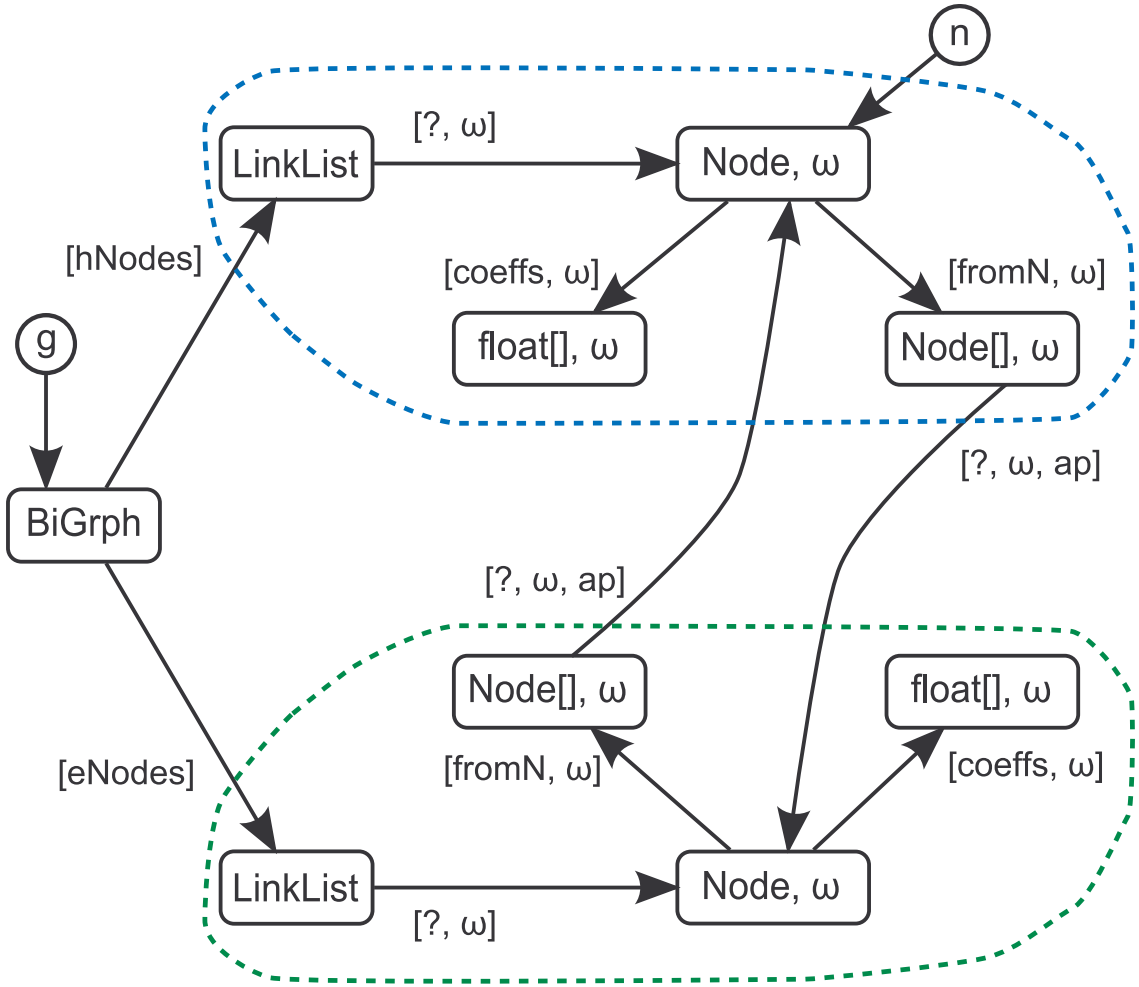
$$(\underbrace{\{v_0 = (\text{data}{:}p_0, \text{next}{:}p_1)\}}_{\text{shape}}, \underbrace{\{\{v_0, p_0\}, \{v_o, p_1\}\}}_{\text{set sharing}})$$

A statement like $v_1 = v_0\,.\,\texttt{data}$ will result in a final abstract state:

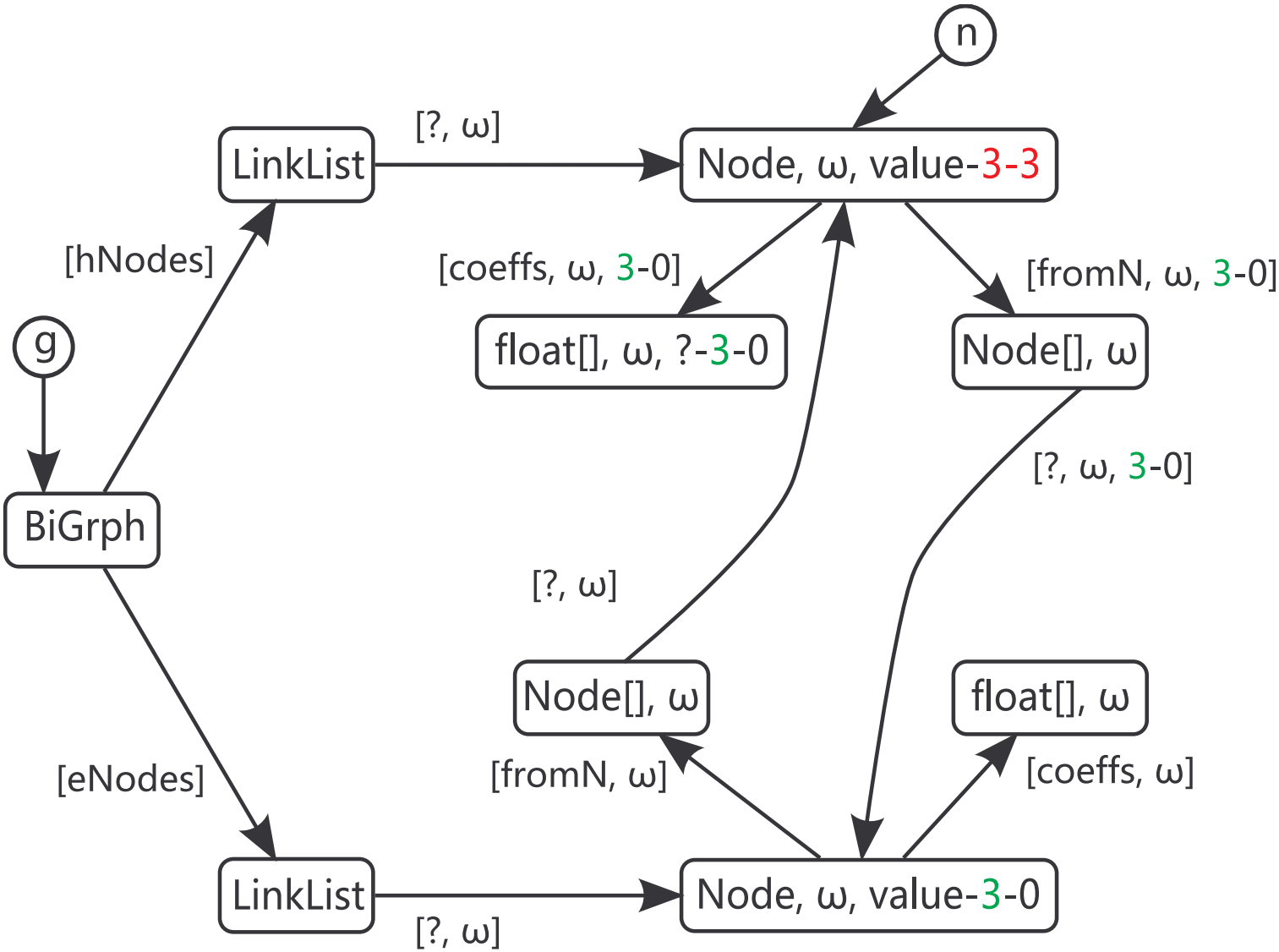
$$(\{v_0 = (\text{data}{:}p_0, \text{next}{:}p_1)\}, \{\{v_0, v_1, p_0\}, \{v_o, p_1\}\})$$

- General support in framework for domain combinations.

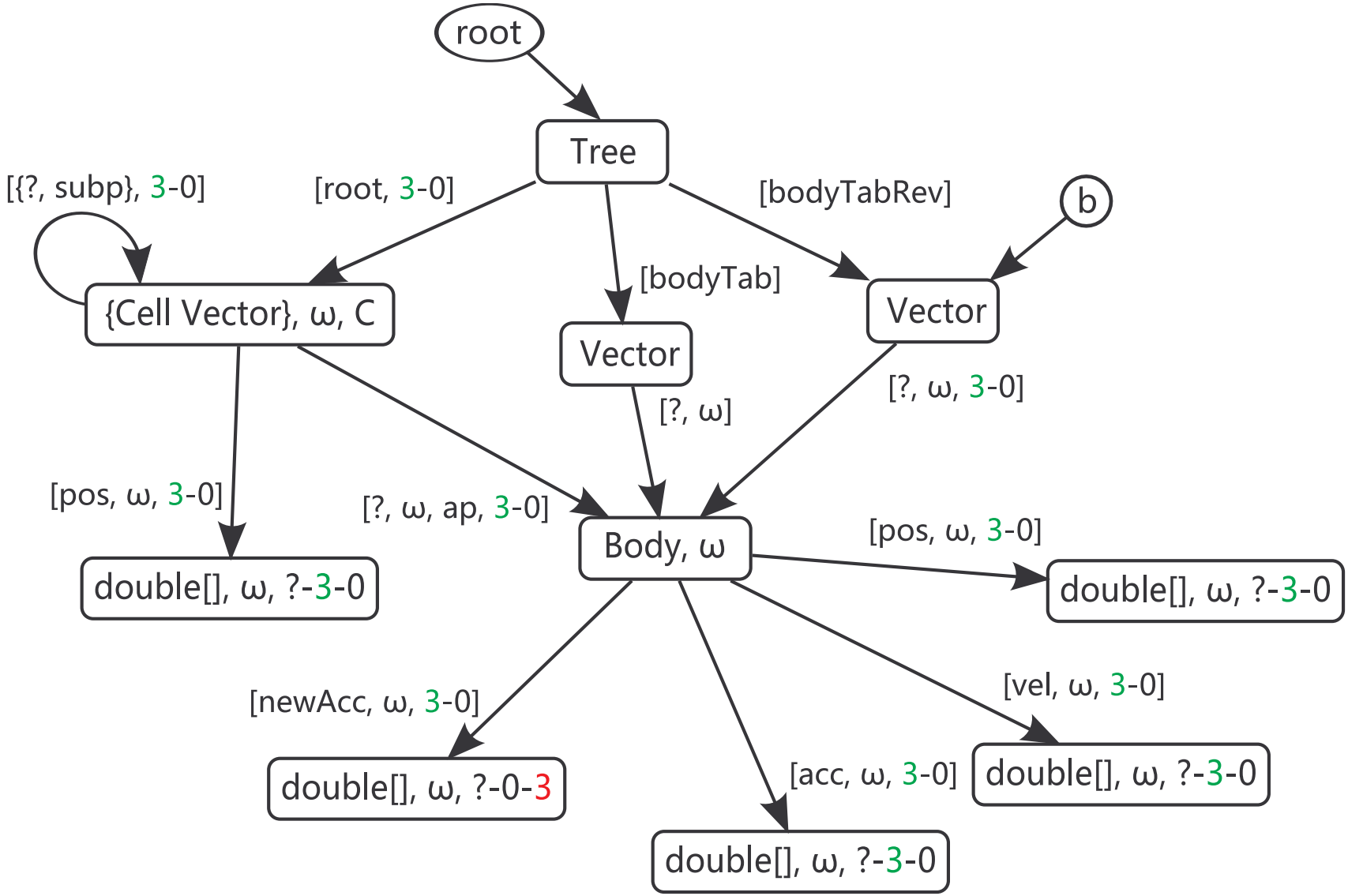
Example: Shape / Heap Dependency (em3d, regions)



Example: Shape / Heap Dependency (em3d, r-w deps)



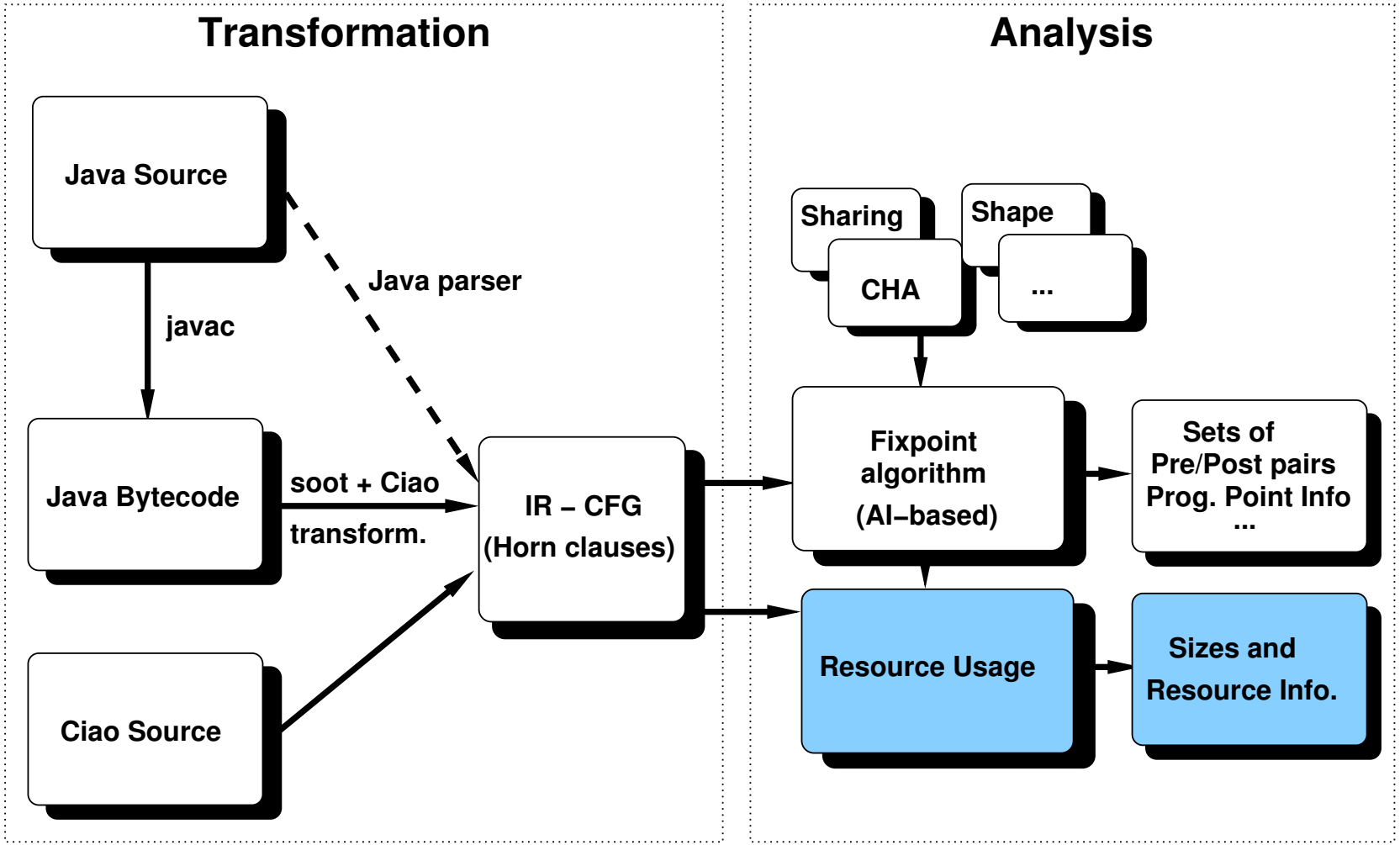
Example: Shape / Heap Dependency (bh, r-w deps)



Example: Shape / Heap Dependency (JOlden,SPECjvm98)

Benchmark	LOC	Classes	Methods	Time	Shape	RW Dep
bisort	560	36	348	0.26s	Y	Y
mst	668	52	485	0.12s	Y	Y
tsp	910	42	429	0.15s	Y	Y
em3d	1103	56	488	0.31s	Y	Y
perimeter	1114	44	381	0.91s	P	N
health	1269	59	534	1.25s	Y	Y
voronoi	1324	58	549	1.80s	Y	Y
power	1752	57	520	0.36s	Y	Y
bh	2304	61	576	1.84s	P	Y
db	1985	68	562	1.42s	Y	Y
logic	3960	72	620	48.26s	P	Y
raytrace	5809	63	506	37.09s	Y	Y

Resource Analyzers



[DLH90, LGHD94, LGHD96, DLGHL94, DLGHL97, NMLGH07, MLGCH08, NMLH08, NMLH09]

Inference of user-defined resource usage

Build analysis which automatically infers upper bounds on the usage that a program makes of a general notion of user-definable resources.

- Examples:
  - Memory, execution time, execution steps, data sizes.
  - Bits sent or received over a socket, SMSs sent or received, accesses to a database, calls to a procedure, files left open, money spent, ..
  - Energy consumed, ...
- Approach:
  1. Programmer defines via *assertions* resource-related properties for basic procedures (e.g., libraries).
  2. System infers the resource usage bounds for rest of program as **functions of input data sizes**.
- Property clearly undecidable → approximation required (**bounds** that are safe and also as accurate as possible).
- Applications: performance debugging and verification, proof carrying code, resource-oriented optimization, ...

[NMLGH07, NMLH09]

Inference of user-defined resource usage

Build analysis which automatically infers upper bounds on the usage that a program makes of a general notion of user-definable resources.

- Examples:
  - Memory, execution time, execution steps, data sizes.
  - Bits sent or received over a socket, SMSs sent or received, accesses to a database, calls to a procedure, files left open, money spent, ..
  - Energy consumed, ...
- Approach:
  1. Programmer defines via *assertions* resource-related properties for basic procedures (e.g., libraries).
  2. System infers the resource usage bounds for rest of program as **functions of input data sizes**.
- Property clearly undecidable → approximation required (**bounds** that are safe and also as accurate as possible).
- Applications: performance debugging and verification, proof carrying code, resource-oriented optimization, ...

[NMLGH07, NMLH09]

Inference of user-defined resource usage

Build analysis which automatically infers upper bounds on the usage that a program makes of a general notion of user-definable resources.

- Examples:
  - Memory, execution time, execution steps, data sizes.
  - Bits sent or received over a socket, SMSs sent or received, accesses to a database, calls to a procedure, files left open, money spent, ..
  - Energy consumed, ...
- Approach:
  1. Programmer defines via *assertions* resource-related properties for basic procedures (e.g., libraries).
  2. System infers the resource usage bounds for rest of program as **functions of input data sizes**.
- Property clearly undecidable → approximation required (**bounds** that are safe and also as accurate as possible).
- Applications: performance debugging and verification, proof carrying code, resource-oriented optimization, ...

[NMLGH07, NMLH09]

Inference of user-defined resource usage

Build analysis which automatically infers upper bounds on the usage that a program makes of a general notion of user-definable resources.

- Examples:
  - Memory, execution time, execution steps, data sizes.
  - Bits sent or received over a socket, SMSs sent or received, accesses to a database, calls to a procedure, files left open, money spent, ..
  - Energy consumed, ...
- Approach:
  1. Programmer defines via *assertions* resource-related properties for basic procedures (e.g., libraries).
  2. System infers the resource usage bounds for rest of program as **functions of input data sizes**.
- Property clearly undecidable $\rightarrow$ approximation required (**bounds** that are safe and also as accurate as possible).
- Applications: performance debugging and verification, proof carrying code, resource-oriented optimization, ...

[NMLGH07, NMLH09]

User-definable aspects of the analysis

- A *cost model* defines an *upper/lower bound cost* for primitive operations (e.g., methods, bytecode instructions).
  - Provided by the user, via *the assertion language*.

```
@Cost("cents","2*size(data)")
public native void Stream.send(java.lang.String data);
```

  - Some predefined in system libraries.

  For platform-dependent resources such as execution time or energy consumption model needs to consider low level factors.
- Assertions:
  - Also used to provide other inputs to the resource analysis such as argument sizes, size metrics, etc. if needed.
  - Also allow improving the accuracy and scalability of the system.
  - Output of resource analysis also expressed via assertions.
  - Used additionally to state resource-related specifications which allows finding bugs, verifying, certifying, etc.

User-definable aspects of the analysis

- A *cost model* defines an *upper/lower bound cost* for primitive operations (e.g., methods, bytecode instructions).
  - Provided by the user, via *the assertion language*.

```
@Cost("cents","2*size(data)")
public native void Stream.send(java.lang.String data);
```

  - Some predefined in system libraries.

  For platform-dependent resources such as execution time or energy consumption model needs to consider low level factors.
- Assertions:
  - Also used to provide other inputs to the resource analysis such as argument sizes, size metrics, etc. if needed.
  - Also allow improving the accuracy and scalability of the system.
  - Output of resource analysis also expressed via assertions.
  - Used additionally to state resource-related specifications which allows finding bugs, verifying, certifying, etc.

Overview of the Analysis

1. Pre-analysis phase using the fixpoint analyzers:
   - Class hierarchy analysis simplifies CFG and improves overall precision.
   - Sharing analysis for correctness (conservative: only when there is no sharing among data structures –currently limited to acyclic).
   - *Determinacy* information inferred and used to obtain tighter bounds.
   - *Non-failure* (no exceptions) inferred for non-trivial lower bounds.
2. Set up recurrence equations representing the size of each output argument as a function of the input data sizes.
   - Data dependency graphs determine *relative* sizes of variable contents. (Size measures are derived from inferred shape information.)
3. Compute upper bounds to the solutions of these recurrence equations to obtain bounds on output argument **sizes**.
   - We have a simple recurrence solver, although the system can easily interface with tools like Parma, PUBS, Mathematica, Matlab, etc.
4. Use the size information to set up recurrence equations representing the computational cost of each block and compute upper bounds to their solutions to obtain **resource usage**.

Overview of the Analysis

1. Pre-analysis phase using the fixpoint analyzers:
   - Class hierarchy analysis simplifies CFG and improves overall precision.
   - Sharing analysis for correctness (conservative: only when there is no sharing among data structures –currently limited to acyclic).
   - *Determinacy* information inferred and used to obtain tighter bounds.
   - *Non-failure* (no exceptions) inferred for non-trivial lower bounds.
2. Set up recurrence equations representing the size of each output argument as a function of the input data sizes.
   - Data dependency graphs determine *relative* sizes of variable contents. (Size measures are derived from inferred shape information.)
3. Compute upper bounds to the solutions of these recurrence equations to obtain bounds on output argument **sizes**.
   - We have a simple recurrence solver, although the system can easily interface with tools like Parma, PUBS, Mathematica, Matlab, etc.
4. Use the size information to set up recurrence equations representing the computational cost of each block and compute upper bounds to their solutions to obtain **resource usage**.

Overview of the Analysis

1. Pre-analysis phase using the fixpoint analyzers:
   - Class hierarchy analysis simplifies CFG and improves overall precision.
   - Sharing analysis for correctness (conservative: only when there is no sharing among data structures –currently limited to acyclic).
   - *Determinacy* information inferred and used to obtain tighter bounds.
   - *Non-failure* (no exceptions) inferred for non-trivial lower bounds.
2. Set up recurrence equations representing the size of each output argument as a function of the input data sizes.
   - Data dependency graphs determine *relative* sizes of variable contents. (Size measures are derived from inferred shape information.)
3. Compute upper bounds to the solutions of these recurrence equations to obtain bounds on output argument **sizes**.
   - We have a simple recurrence solver, although the system can easily interface with tools like Parma, PUBS, Mathematica, Matlab, etc.
4. Use the size information to set up recurrence equations representing the computational cost of each block and compute upper bounds to their solutions to obtain **resource usage**.

Overview of the Analysis

1. Pre-analysis phase using the fixpoint analyzers:
   - Class hierarchy analysis simplifies CFG and improves overall precision.
   - Sharing analysis for correctness (conservative: only when there is no sharing among data structures –currently limited to acyclic).
   - *Determinacy* information inferred and used to obtain tighter bounds.
   - *Non-failure* (no exceptions) inferred for non-trivial lower bounds.
2. Set up recurrence equations representing the size of each output argument as a function of the input data sizes.
   - Data dependency graphs determine *relative* sizes of variable contents. (Size measures are derived from inferred shape information.)
3. Compute upper bounds to the solutions of these recurrence equations to obtain bounds on output argument **sizes**.
   - We have a simple recurrence solver, although the system can easily interface with tools like Parma, PUBS, Mathematica, Matlab, etc.
4. Use the size information to set up recurrence equations representing the computational cost of each block and compute upper bounds to their solutions to obtain **resource usage**.

Example: sending SMSs

```
public class CellPhone {

 void sendSms(SmsPacket smsPk,
              Encoder enc,
              Stream stm) {
 if (smsPk != null) {
  stm.send(
        enc.format(smsPk.sms));
  sendSms(smsPk.next,enc,stm);
 }}}

class SmsPacket{
   String sms;
   SmsPacket next;
}

abstract class Stream{
   @Cost({"cents","2*size(data)"})
   native void send(String data);
}
```

```
interface Encoder{
   String format(String data);
   }

class TrimEncoder implements Encoder{
   @Cost({"cents","0"})
   @Size("size(ret)<=size(s)")
   public String format(String s){
     return s.trim();
   }}

class UnicodeEncoder implements Encoder{
   @Cost({"cents","0"})
   @Size("size(ret)<=6*size(s)")
   public String format(String s){
     return java.net.URLEncoder.encode(s)
   }}
```

Example (I)

1. System takes by default size of input data: $size(\texttt{smsPk}) = n$.
   - Result will be parametric on this.
2. The number of characters *sent* depends on the formatting done by the different encoders:
   - The user indicates that the encoding in `TrimEncoder` results in a smaller or equal (output) string.

```
class TrimEncoder implements Encoder{
    @Size("size(ret)<=size(s)")
    public String format(String s){
```

   - And that the result of `UnicodeEncoder` can be up to 6 times larger (\uxxxx) than the one received.

```
class UnicodeEncoder implements Encoder{
    @Size("size(ret)<=6*size(s)")
    public String format(String s){
```

Example (II)

3. After setting up and solving the size equations the system obtains that the upper bound on the number of characters sent is:

$$max(6,1) * n = 6 * n = 6 * size(\texttt{smsPk})$$

4. The analysis establishes then (cost) recurrences for every method:

$$\begin{array}{lcl} Cost_{sendSms}(r0,0,r2,r3) & = & 0 \\ Cost_{sendSms}(r0,r1,r2,r3) & = & cost\ of\ sending\ a\ char \times Cost_{sendSms}(r0,r1-1,r2,r3) \end{array}$$

where $r0$,$r1$,$r2$, and $r3$ represent the size of This, SmsPk, enc, and stm, respectively.

5. Given that we are charged 2 cents per character sent:

```
@Cost({"cents","2*size(data)"})
native void send(String data);
```

$$\begin{array}{lcl} Cost_{sendSms}(r0,0,r2,r3) & = & 0 \\ Cost_{sendSms}(r0,r1,r2,r3) & = & 2\times \underbrace{6 \times (r1-1)}_{character\ size} \times Cost_{sendSms}(r0,r1-1,r2,r3) \end{array}$$

and the total cost of the `sendSMS` method is $6 \times r1^2 - 6 \times r1$ cents.

Some results (Java)

Program	Resource(s)	t	Resource Usage Func.	/ Metric
BST	Heap usage	367	$O(2^n)$	$n \equiv$ tree depth
CellPhone	SMS monetary cost	386	$O(n^2)$	$n \equiv$ packets length
Client	Bytes received and	527	$O(n)$	$n \equiv$ stream length
	bandwidth required		$O(1)$	—
Dhrystone	Energy consumption	759	$O(n)$	$n \equiv$ int value
Divbytwo	Stack usage	219	$O(log_2(n))$	$n \equiv$ int value
Files	Files left open and	649	$O(n)$	$n \equiv$ number of files
	Data stored		$O(n \times m)$	$m \equiv$ stream length
Join	DB accesses	460	$O(n \times m)$	$n, m \equiv$ table records
Screen	Screen width	536	$O(n)$	$n \equiv$ stream length

- Different complexity functions, resources, types of loops/recursion, etc.

Some results (Ciao)

Program	Resource	Usage Function	Metrics	Time
client	"bits received"	$\lambda x.8 \cdot x$	length	186
color_map	"unifications"	39066	size	176
copy_files	"files left open"	$\lambda x.x$	length	180
eight_queen	"queens movements"	19173961	length	304
eval_polynom	"FPU usage"	$\lambda x.2.5x$	length	44
fib	"arith. operations"	$\lambda x.2.17 \cdot 1.61^x + 0.82 \cdot (-0.61)^x - 3$	value	116
grammar	"phrases"	24	length/size	227
hanoi	"disk movements"	$\lambda x.2^x - 1$	value	100
insert_stores	"accesses Stores" "insertions Stores"	$\lambda n, m.n + k$ $\lambda n, m.n$	length	292
perm	"WAM instructions"	$\lambda x.(\sum_{i=1}^{x} 18 \cdot x!) + (\sum_{i=1}^{x} 14 \cdot \frac{x!}{i}) + 4 \cdot x!$	length	98
power_set	"output elements"	$\lambda x.\frac{1}{2} \cdot 2^{x+1}$	length	119
qsort	"lists parallelized"	$\lambda x.4 \cdot 2^x - 2x - 4$	length	144
send_files	"bytes read"	$\lambda x, y.x \cdot y$	length/size	179
subst_exp	"replacements"	$\lambda x, y.2xy + 2y$	size/length	153
zebra	"resolution steps"	302328442957130 61	size	292

Interesting Resource: Execution Time

- Important: e.g., verification of real-time constraints.
- Very hard in current architectures, (e.g., worst-case cache behavior).
  - Certainly feasible in simple processors and with caches turned off.
  - Our approach is *complementary* to accurate WCET models, which consider cache behavior, pipeline state, etc. (inputs to us).
- Approach:
  - Obtain timing model of abstract machine instructions through a one-time profiling phase (results provided as assertions).
    - Includes fitting constants in a function if the execution time depends on the argument's properties.
  - Static cost analysis phase which infers a function which returns (bounds on) the execution time of program for given input data sizes.

[MLGCH08]

Interesting Resource: Execution Time

- Important: e.g., verification of real-time constraints.
- Very hard in current architectures, (e.g., worst-case cache behavior).
  - Certainly feasible in simple processors and with caches turned off.
  - Our approach is *complementary* to accurate WCET models, which consider cache behavior, pipeline state, etc. (inputs to us).
- Approach:
  - Obtain timing model of abstract machine instructions through a one-time profiling phase (results provided as assertions).
    - Includes fitting constants in a function if the execution time depends on the argument's properties.
  - Static cost analysis phase which infers a function which returns (bounds on) the execution time of program for given input data sizes.

[MLGCH08]

First Phase Output

Cost assertions automatically generated in first phase and stored to make the instruction execution costs available to the static analyzer.

Examples

```
:- true pred unify_variable(A, B): int(A), int(B)
   + (cost(ub, exectime, 667.07),
      cost(lb, exectime, 667.07)).

:- true pred unify_variable(A, B): var(A), gnd(B)
   + (cost(ub, exectime, 233.3),
      cost(lb, exectime, 233.3)).

:- true pred unify_variable(A, B): list(A),list(B)
  + cost(ub, exectime, 271.58+284.34*length(A)).    ...
```

Observed and Estimated Execution Time (Intel)

Pr. No.	Cost. App.	Intel (μs) Est.	Prf.	Obs.	D. %	Pr.D. %
1	E	110	110	113	-2.4	-2.4
2	E	69	69	71	-2.3	-2.3
3	E	1525	1525	1576	-3.3	-3.3
4	E	1501	1501	1589	-5.7	-5.7
5	E	2569	2569	2638	-2.7	-2.7
6	E	1875	1875	2027	-7.8	-7.8
7	E	1868	1868	1931	-3.3	-3.3
8	L	43	68	81	-67.2	-17.8
	U	3414	3569	3640	-6.4	-2.0
9	L	54	79	91	-54.6	-14.8
	U	3414	3694	4011	-16.2	-8.2
10	L	135	142	124	8.6	13.7
	U	7922	2937	2858	120.6	2.7
11	L	216	138	111	72.3	22.5
	U	226	216	162	34.0	29.5

Energy Consumption

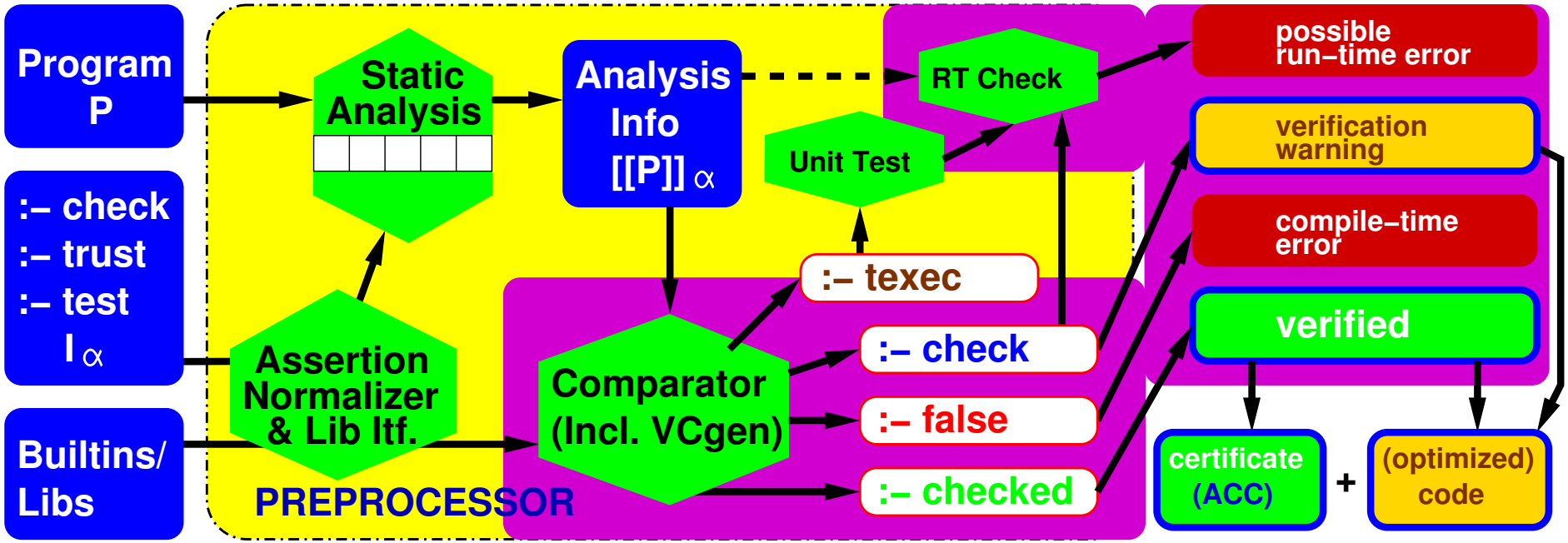
- Energy Consumption Analysis:
  - *Energy consumption model* (available for simple processors): describe upper bound consumption of each bytecode inst. in terms of joules:

Opcode	Inst. Cost in μJ	Mem. Cost in μJ	Total Cost in in μJ
`iadd`	.957860	2.273580	3.23144
`isub`	.957360	2.273580	3.230.94
...	...	...	...

  - Resource analysis generates at compile time equations and returns safe, upper- and lower-bound *energy consumption functions*.

[NMLH08]

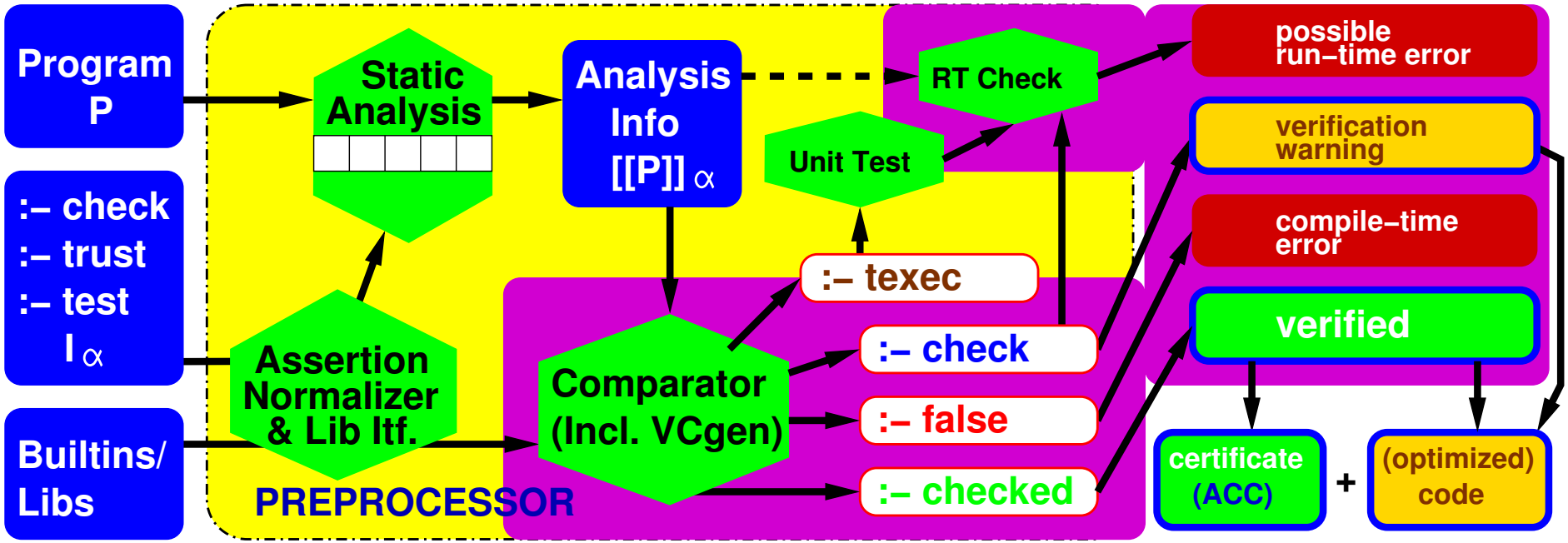
Debugging and Verification



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

[BDD+97, HPB99, PBH00c, PBH00a, HPBLG03, HALGP05, PCPH06, PCPH08, MLGH09]

Debugging and Verification



- *Parts* of assertions not verified statically generate run-time checks.
- Diagnosis (for both static and dynamic errors).
- Comparison not always trivial. E.g., resource debugging/certification:
  - Need to compare functions.
  - "Segmented" answers.

[BDD+97, HPB99, PBH00c, PBH00a, HPBLG03, HALGP05, PCPH06, PCPH08, MLGH09]

Non-trivial: e.g., Resource Debugging / Certification

- Approximated and intended semantics given as *resource usage functions*:

  Monotonic arithmetic functions expressing lower or upper bounds on the resource usage of a predicate depending on input data sizes.

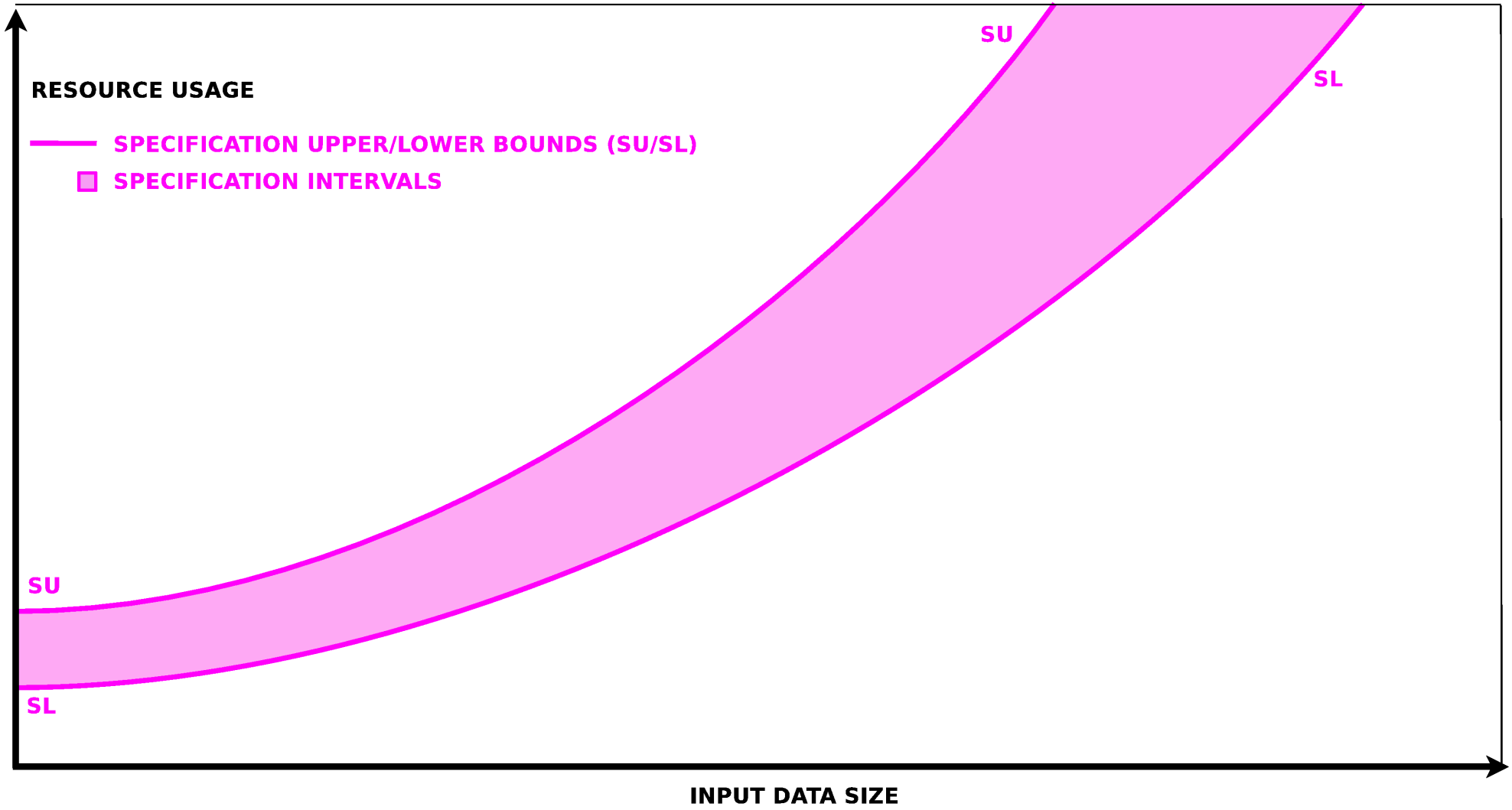
Example of intended semantics (partial specifications):

```
:- check comp nrev(A,B) :   (ground(A), list(A), var(B) )
                      +     (resource(lb, steps, length(A)),
                             resource(ub, steps, 1 + exp(length(A), 2)) .
```

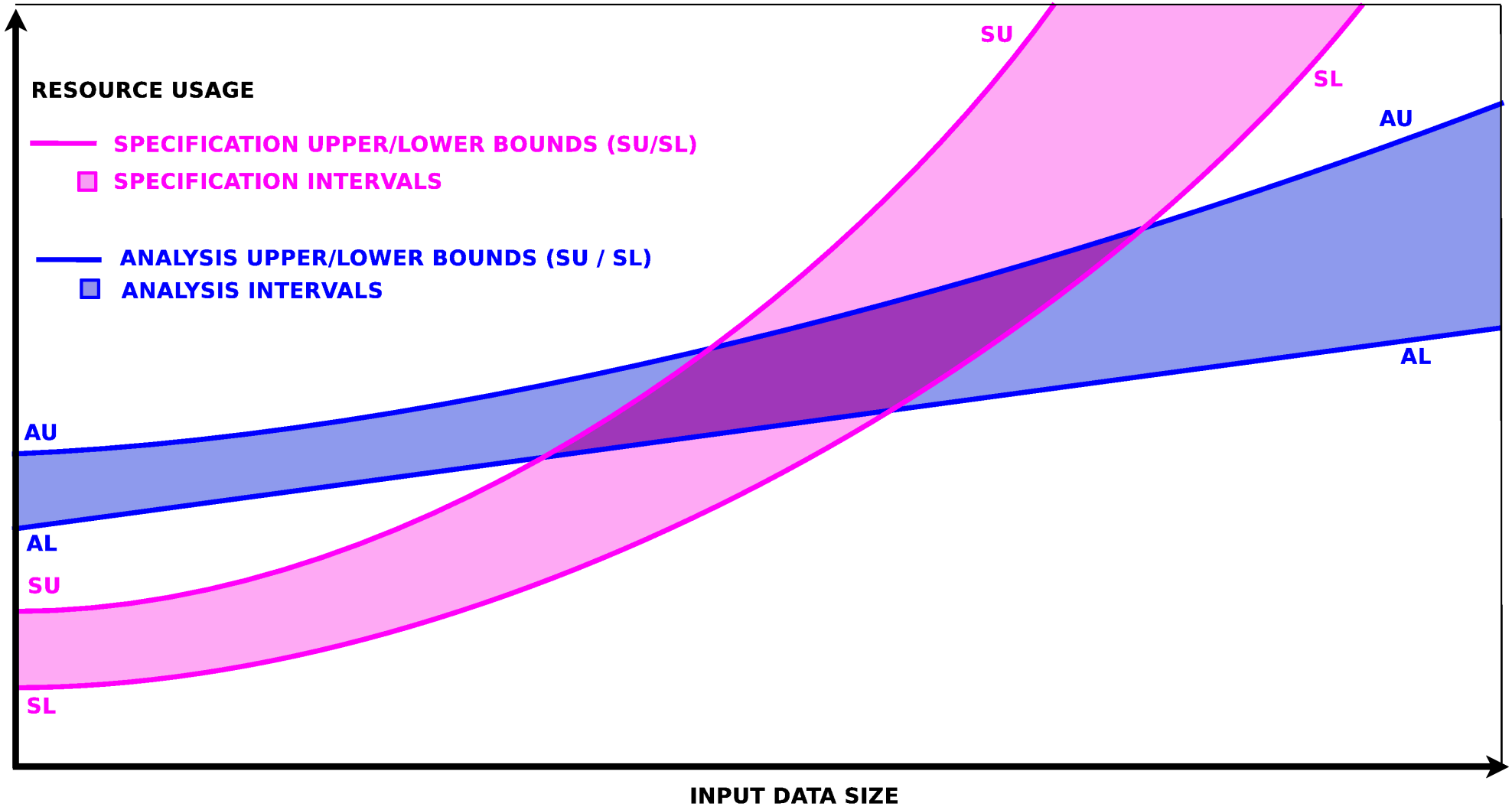
- Number of steps $\in [length(A), 1 + length(A)^2]$.

[HALGP05, LGDB10]

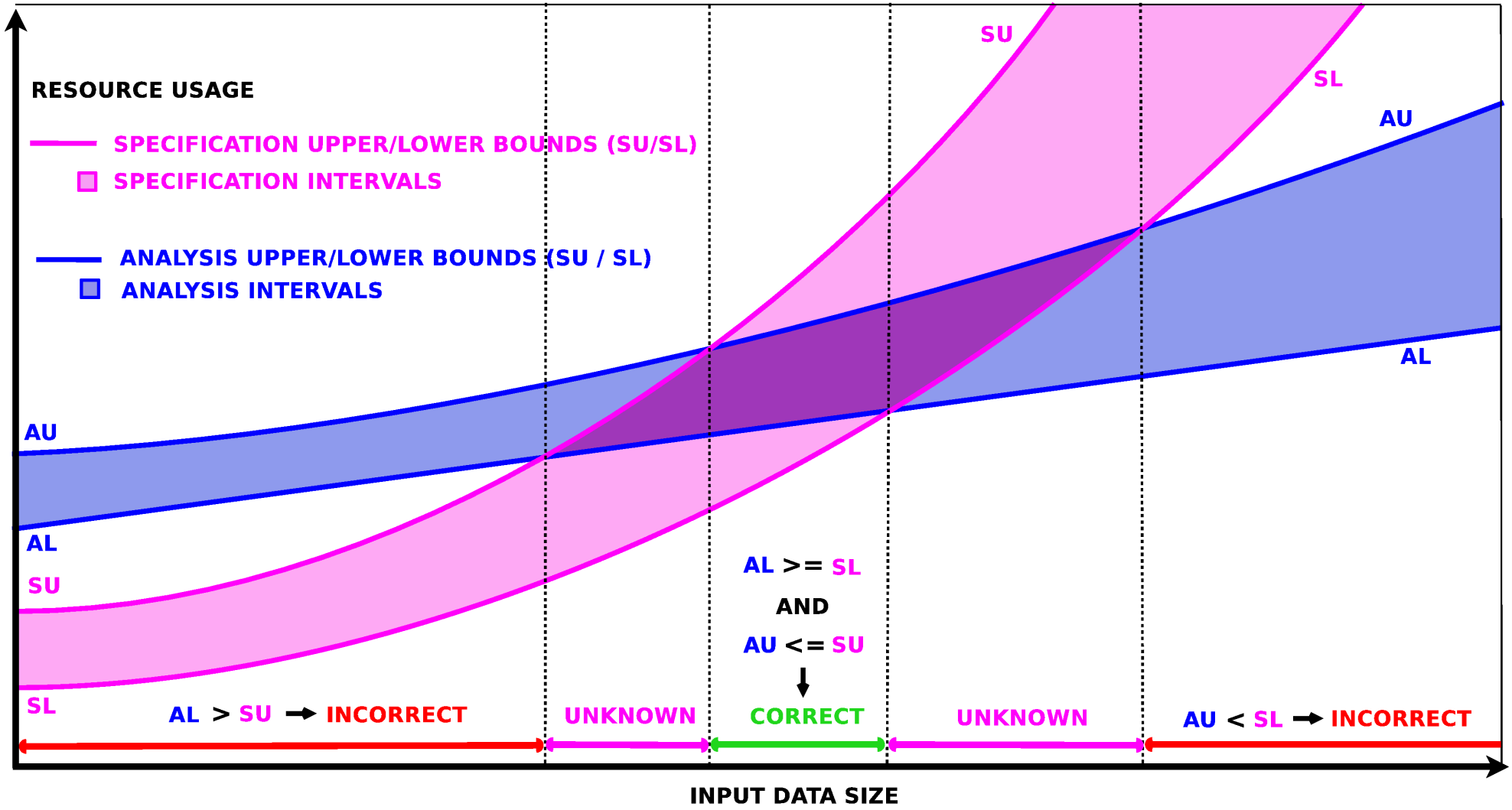
Resource Usage Verification



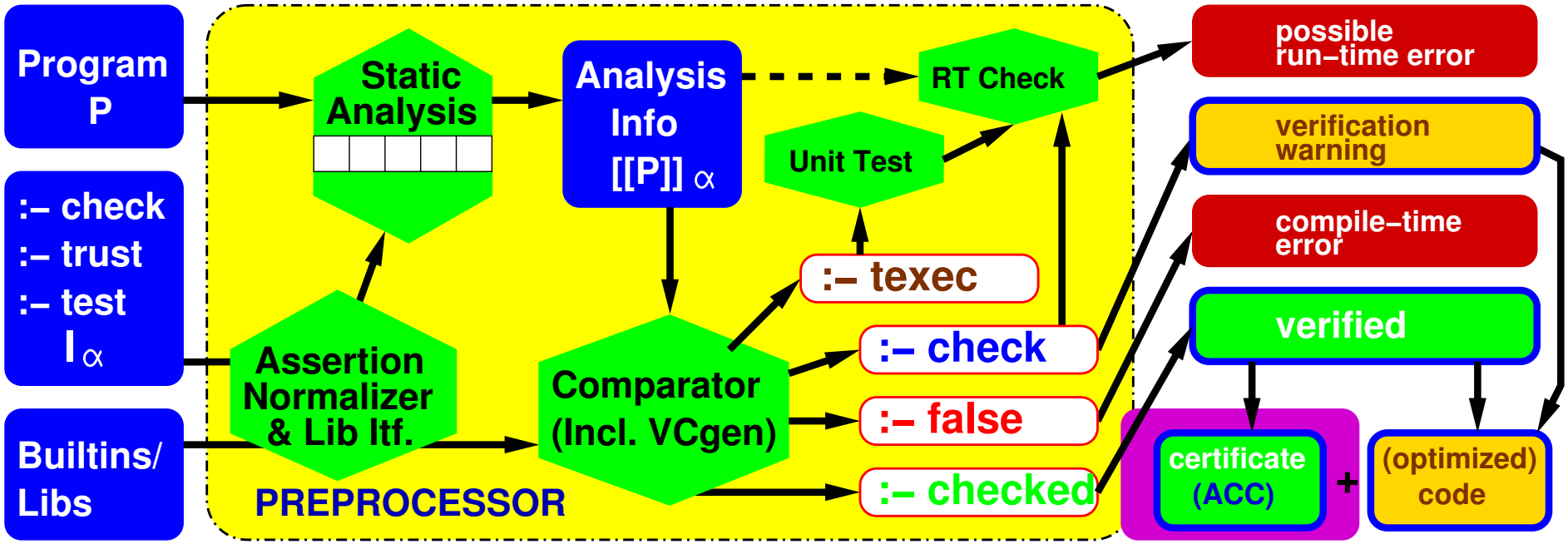
Resource Usage Verification



Resource Usage Verification

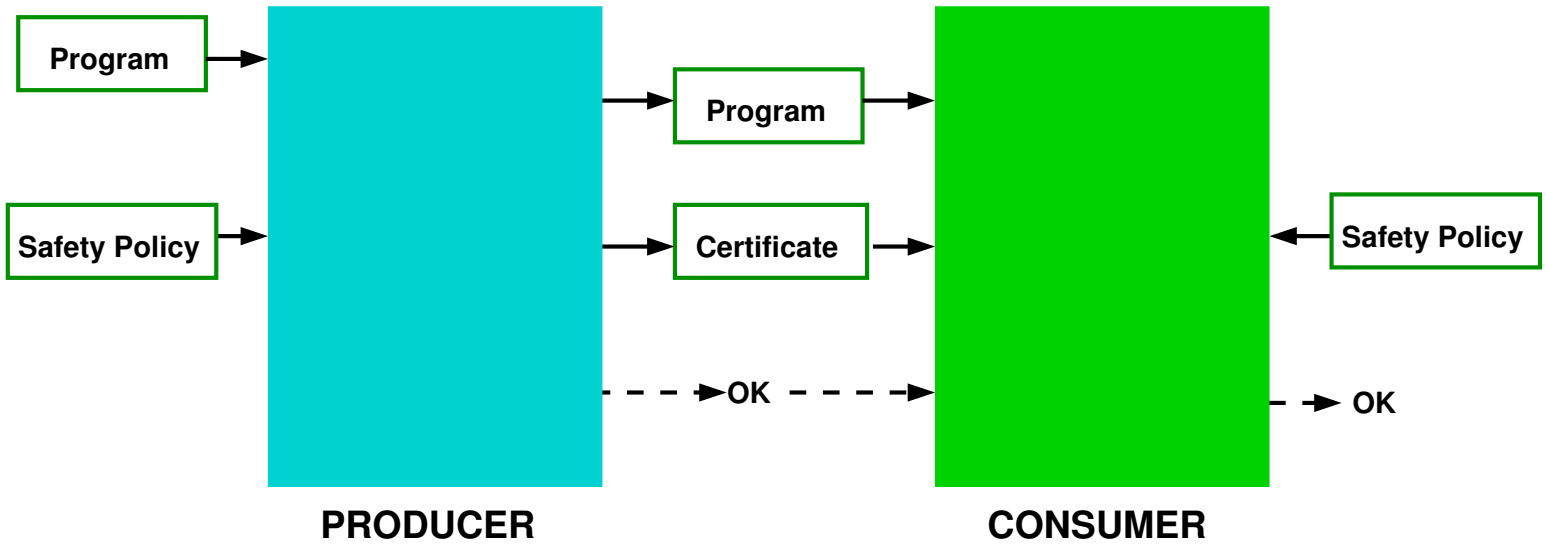


Certification / Abstraction Carrying Code



[APH05, HALGP05, AAPH06]

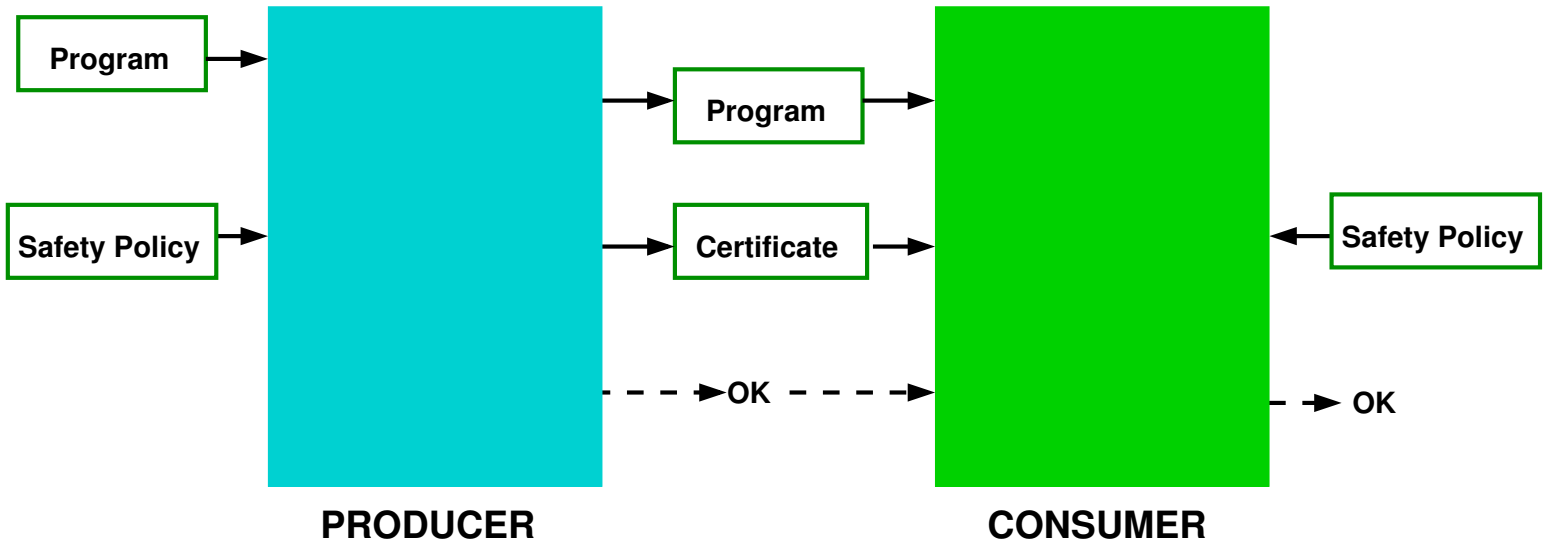
Code Certification and Proof-Carrying Code



Many challenges:

- Generating the certificates automatically.
- Generating minimal certificates.
- Designing *simple, reliable, and efficient checkers* for the certificates.

Code Certification and Proof-Carrying Code

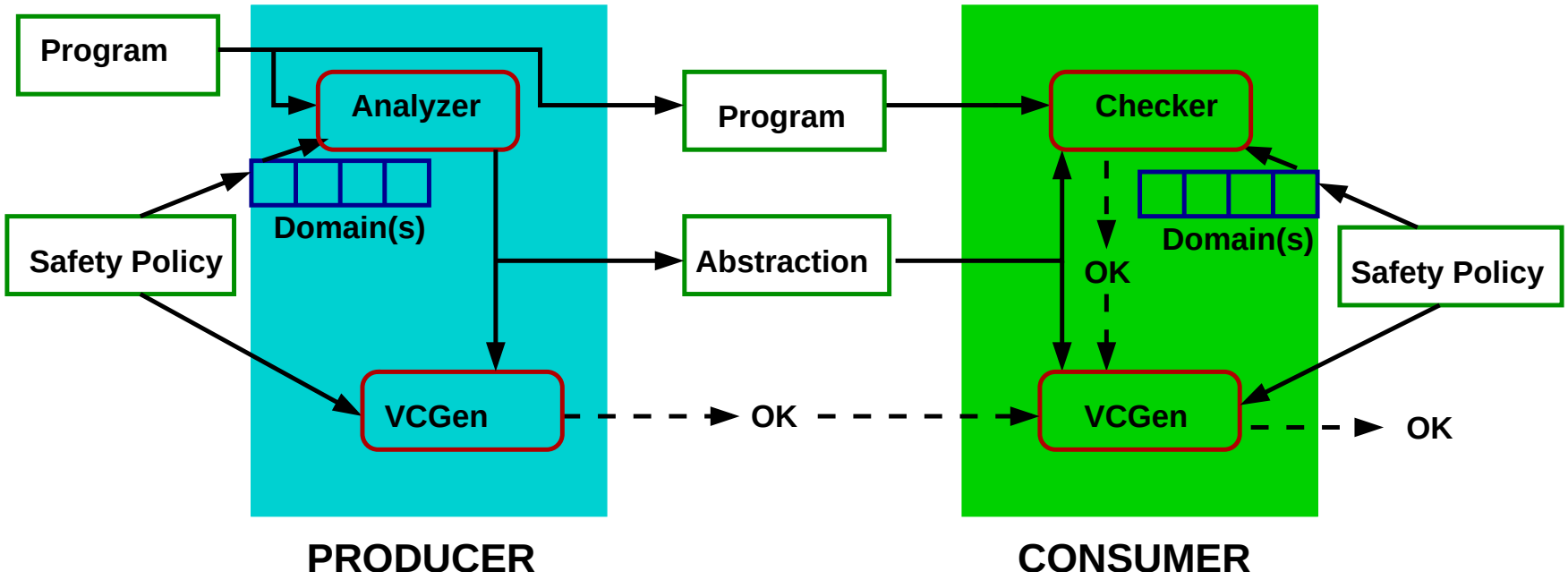


Many challenges:

- Generating the certificates automatically.
- Generating minimal certificates.
- Designing *simple, reliable, and efficient checkers* for the certificates.

Abstraction-based Certification, Abstraction-Carrying Code

- The Abstraction Carrying Code (ACC) scheme [LPAR04]:



$[\![P]\!]_\alpha = \text{Analysis} = \mathbf{lfp}(\mathit{analysis\_step})$

Certificate $\subset [\![P]\!]_\alpha$
Certificate $\rightarrow$ Safety Policy

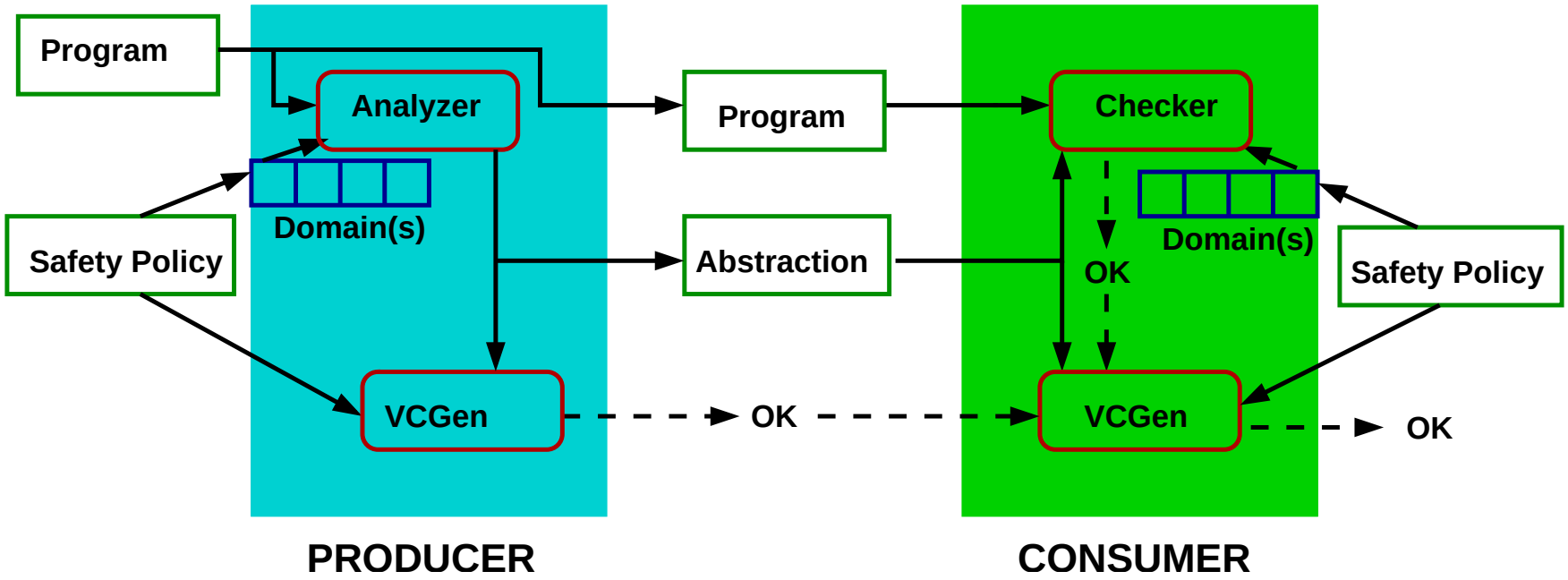
Checker $= \mathit{analysis\_step}$

- Many interesting extensions: reduced certificates, incrementality, ...

[APH05, HALGP05, AAPH06]

Abstraction-based Certification, Abstraction-Carrying Code

- The Abstraction Carrying Code (ACC) scheme [LPAR04]:



$[\![P]\!]_\alpha = \text{Analysis} = \mathbf{lfp}(\mathit{analysis\_step})$

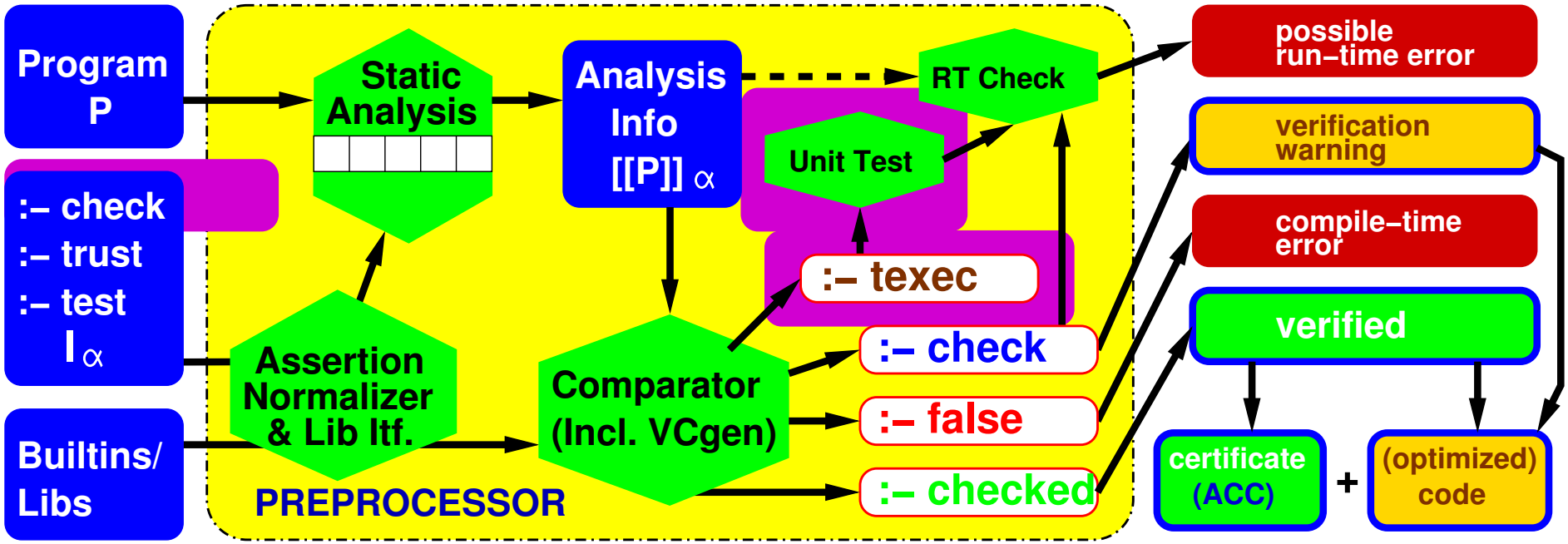
Certificate $\subset [\![P]\!]_\alpha$
Certificate $\rightarrow$ Safety Policy

Checker $= \mathit{analysis\_step}$

- Many interesting extensions: reduced certificates, incrementality, ...

[APH05, HALGP05, AAPH06]

Integration of testing



[MLGH09]

Supporting unit testing

Assertion schema used:

```
:- test Pred[:Precond][=>Postcond][+CompExecProps].
```

Such test assertions translate into:

What needs to be checked (normal assertions):

```
:- check pred Pred [:Precond] [=>Postcond] [+CompProps].
```

What test case needs to be run (test driver):

```
:-       texec   Pred [:Precond] [+Exec-Formula].
```

Many interactions within the integrated framework:

- (Unit) tests are part of the assertion language.
- Parts of unit tests that can be verified at compile-time are deleted.
- Rest of unit testing uses the run-time assertion-checking machinery.
- Unit tests also provide test cases for run-time checks coming from assertions.
  - Assertions checked by unit testing, even if not conceived as tests.

Supporting unit testing

Assertion schema used:

```
:- test Pred[:Precond][=>Postcond][+CompExecProps].
```

Such test assertions translate into:

What needs to be checked (normal assertions):

```
:- check pred Pred [:Precond] [=>Postcond] [+CompProps].
```

What test case needs to be run (test driver):

```
:-       texec   Pred [:Precond] [+Exec-Formula].
```

Many interactions within the integrated framework:

- (Unit) tests are part of the assertion language.
- Parts of unit tests that can be verified at compile-time are deleted.
- Rest of unit testing uses the run-time assertion-checking machinery.
- Unit tests also provide test cases for run-time checks coming from assertions.
  - Assertions checked by unit testing, even if not conceived as tests.

Example assertion and testing output:

```
:- test qsort(A,_) : (A=[1,3,2]) => (ground(B),list(num,B),sorted(B)).
```

```
{In /tmp/qsort.pl
ERROR: (lns 32-32) Run-time check failure in assertion for:
        'qsort:qsort'([3,2],[3,2]).
In *success*, unsatisfied property:
        sorted([3,2]).
ERROR: (lns 33-37) Check failed in 'qsort:qsort'([3,2],[3,2])
ERROR: (lns 33-37) Failed when invocation of 'qsort:qsort'([1,3,2],_)
called 'qsort:qsort'([3,2],_) in its body.
}
```

Example application:

- Coded 976 unit tests for ISO compliance of Ciao `prolog` package.
- Detected large number of previously unknown limitations/errors.
- The tests currently run in under 15 seconds.

Example assertion and testing output:

```
:- test qsort(A,_) :  (A=[1,3,2]) => (ground(B),list(num,B),sorted(B)).
```

```
{In /tmp/qsort.pl
ERROR: (lns 32-32) Run-time check failure in assertion for:
        'qsort:qsort'([3,2],[3,2]).
In *success*, unsatisfied property:
        sorted([3,2]).
ERROR: (lns 33-37) Check failed in 'qsort:qsort'([3,2],[3,2])
ERROR: (lns 33-37) Failed when invocation of 'qsort:qsort'([1,3,2],_)
called 'qsort:qsort'([3,2],_) in its body.
}
```

Example application:

- Coded 976 unit tests for ISO compliance of Ciao `prolog` package.
- Detected large number of previously unknown limitations/errors.
- The tests currently run in under 15 seconds.

Example assertion and testing output:

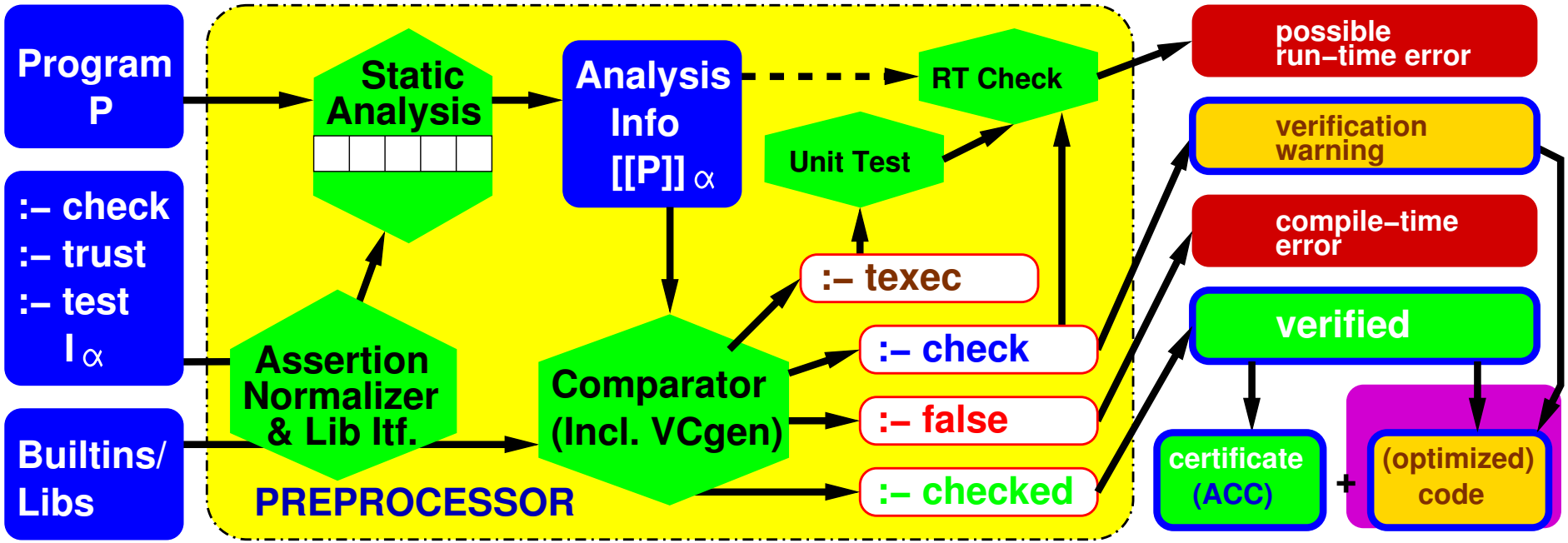
```
:- test qsort(A,_) :  (A=[1,3,2]) => (ground(B),list(num,B),sorted(B)).
```

```
{In /tmp/qsort.pl
ERROR: (lns 32-32) Run-time check failure in assertion for:
        'qsort:qsort'([3,2],[3,2]).
In *success*, unsatisfied property:
        sorted([3,2]).
ERROR: (lns 33-37) Check failed in 'qsort:qsort'([3,2],[3,2])
ERROR: (lns 33-37) Failed when invocation of 'qsort:qsort'([1,3,2],_)
called 'qsort:qsort'([3,2],_) in its body.
}
```

Example application:

- Coded 976 unit tests for ISO compliance of Ciao `prolog` package.
- Detected large number of previously unknown limitations/errors.
- The tests currently run in under 15 seconds.

Optimizations and Parallelization



[GH91, PH97, PHG99, PAH06] [PH99, MBdlBH99, BGH99, CCH08, MKSH08] [MCH04, CMM+06]

Also, optimizations

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

    $\rightarrow$ obtaining close-to-C performance for declarative languages (Ciao).

  - Parallelization.

- But this is a topic for other talks...

[GH91, PH97, PHG99, PAH06] [MCH04, CMM+06]

Parallelization: Motivation

- Power limits frequency... but available area takes us to tera-device: 1000 billion devices by 2020.
- Move from superscalar to multicore: large (100+) numbers of (possibly less complex) (possibly slower) (possibly heterogeneous) cores.
- Novel performance guarantees and architectural contracts (e.g., the memory model) that may not stabilize for several generations.
- Programmability is a huge challenge.

[PH99, MBdlBH99, BGH99, PH97, PH03, CCH08, MKSH08]

(Semi-)Automatic Parallelization and Verification

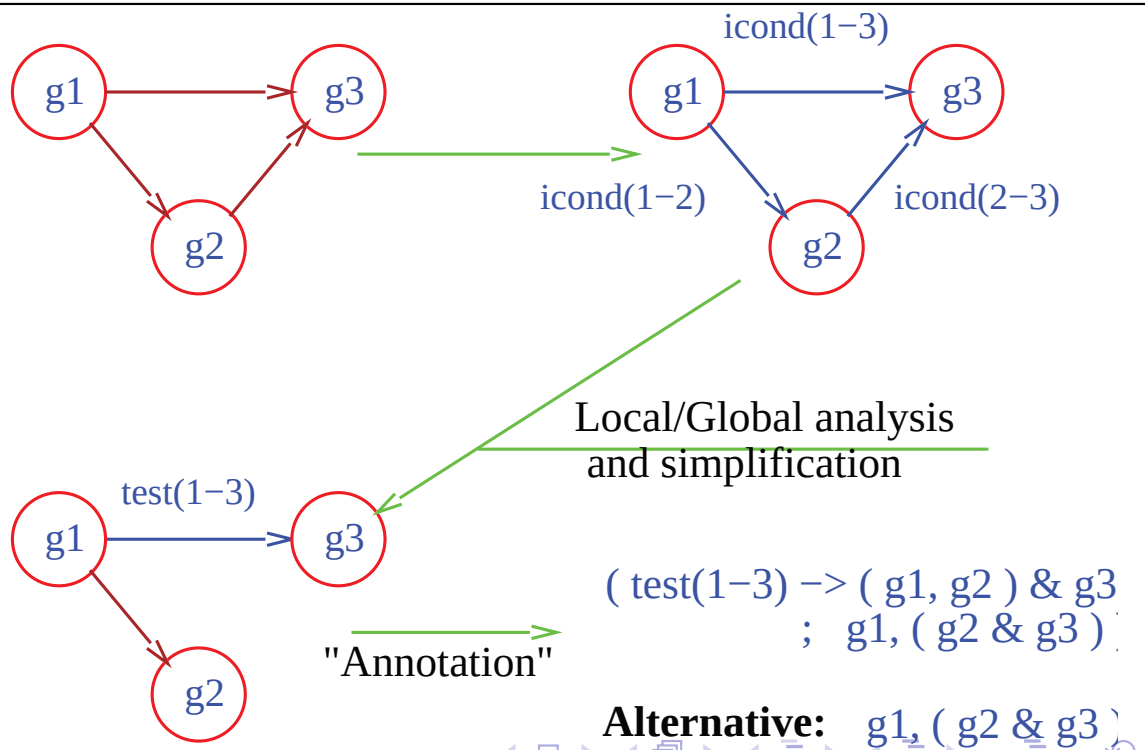
- Automatic parallelization of traditional languages.
  - Limited because of serial nature but still needs to be done:
    - Large number of existing programs.
    - A paradigm shift will take time.
  - Some success for loops. Now, some more progress in dealing with:
    - Irregular control.
    - Complex, dynamically managed data structures.

    Major advances in technology:
    - Aliasing analysis, heap shape analysis, heap dependence tracking, ...
    - Taking into account (new) memory models (coherence models, transactional, parallel garbage collection, ...).
- Compiler-*aided parallelization* and *verification* of parallel programs.
- *Automatic parallelization of less serial (more declarative) languages*.
  - Significant experience from declarative languages needs to be carried over to these new paradigms (see later).

Parallelization Process

- Conditional dependency graph (of some code segment, e.g., a clause):
  - Vertices: possible tasks (statements, calls,...),
  - Edges: possible dependencies (labels: conds. needed for independence).
- Local or global analysis used to reduce/remove checks in the edges.
- Annotation process converts graph into parallel expressions in source.

```
foo(...)   :-
      g1(...),
      g2(...),
      g3(...).
```



Automatic Program Parallelization (Contd.)

- *Example:*

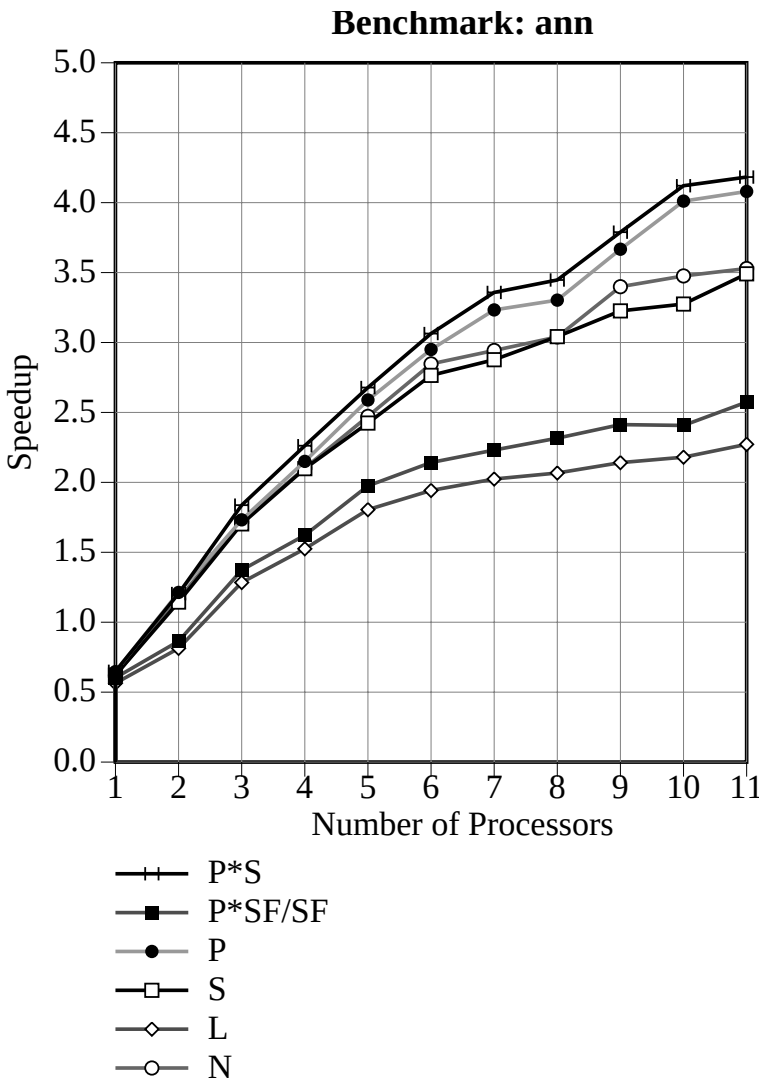
```
qs([X|L],R) :- part(L,X,L1,L2),
               qs(L2,R2), qs(L1,R1),
               app(R1,[X|R2],R).
```

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

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

Global analysis would eliminate the `indep(L1,L2)` check.

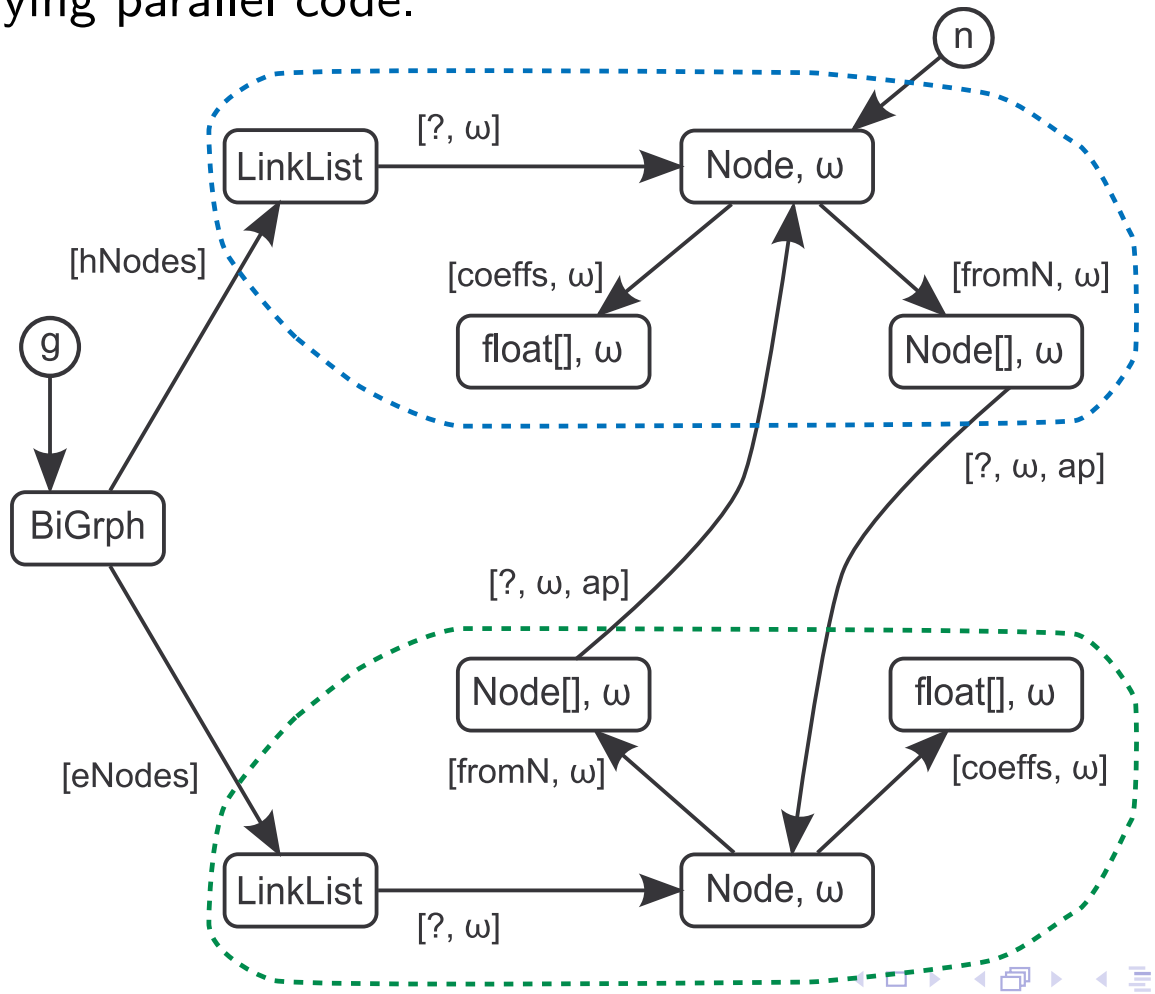
Some Speedups (for different analysis abstract domains)



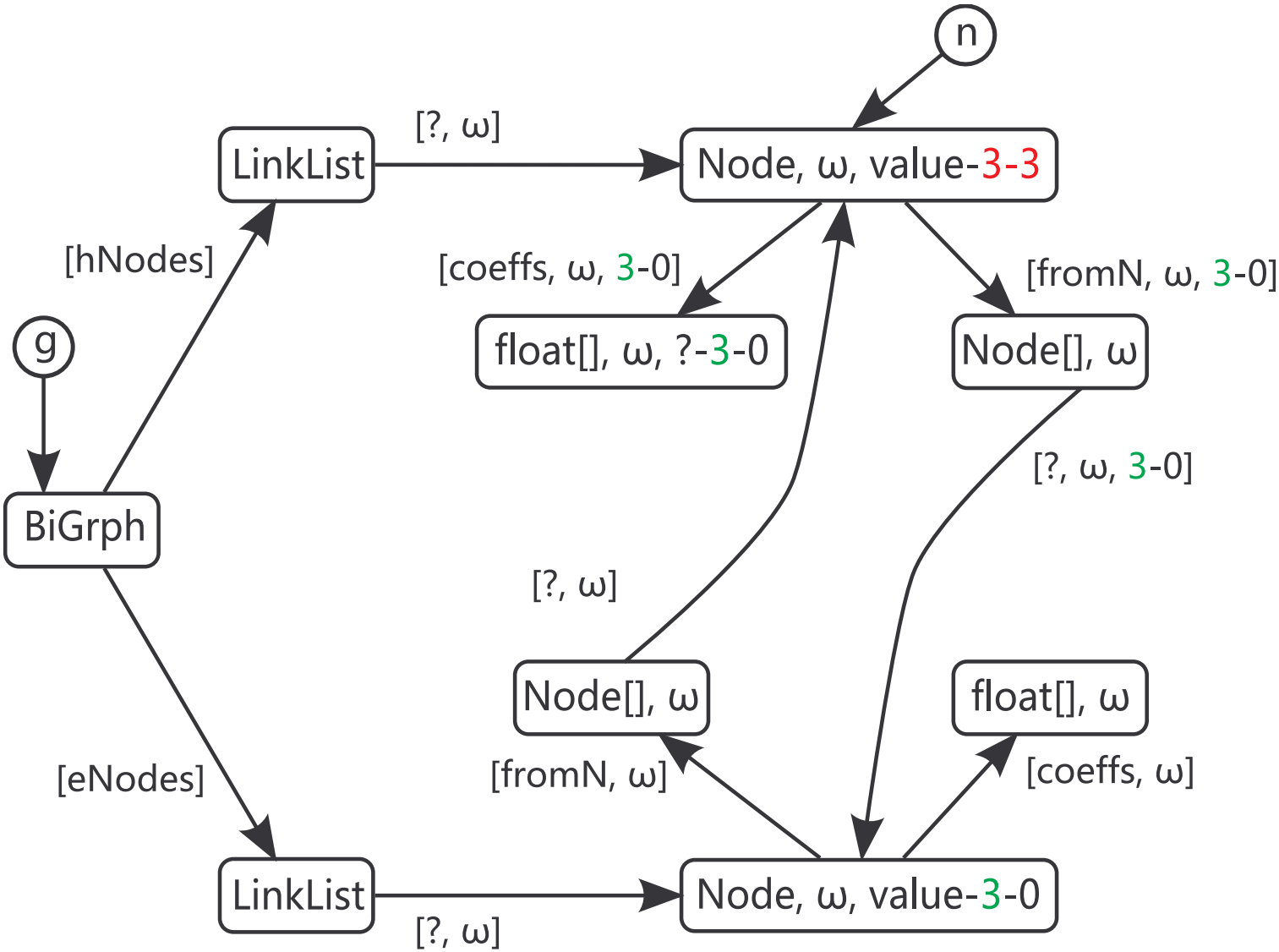
The parallelizer, self-parallelized.

Example: Heap Dependency Analysis (em3d, regions)

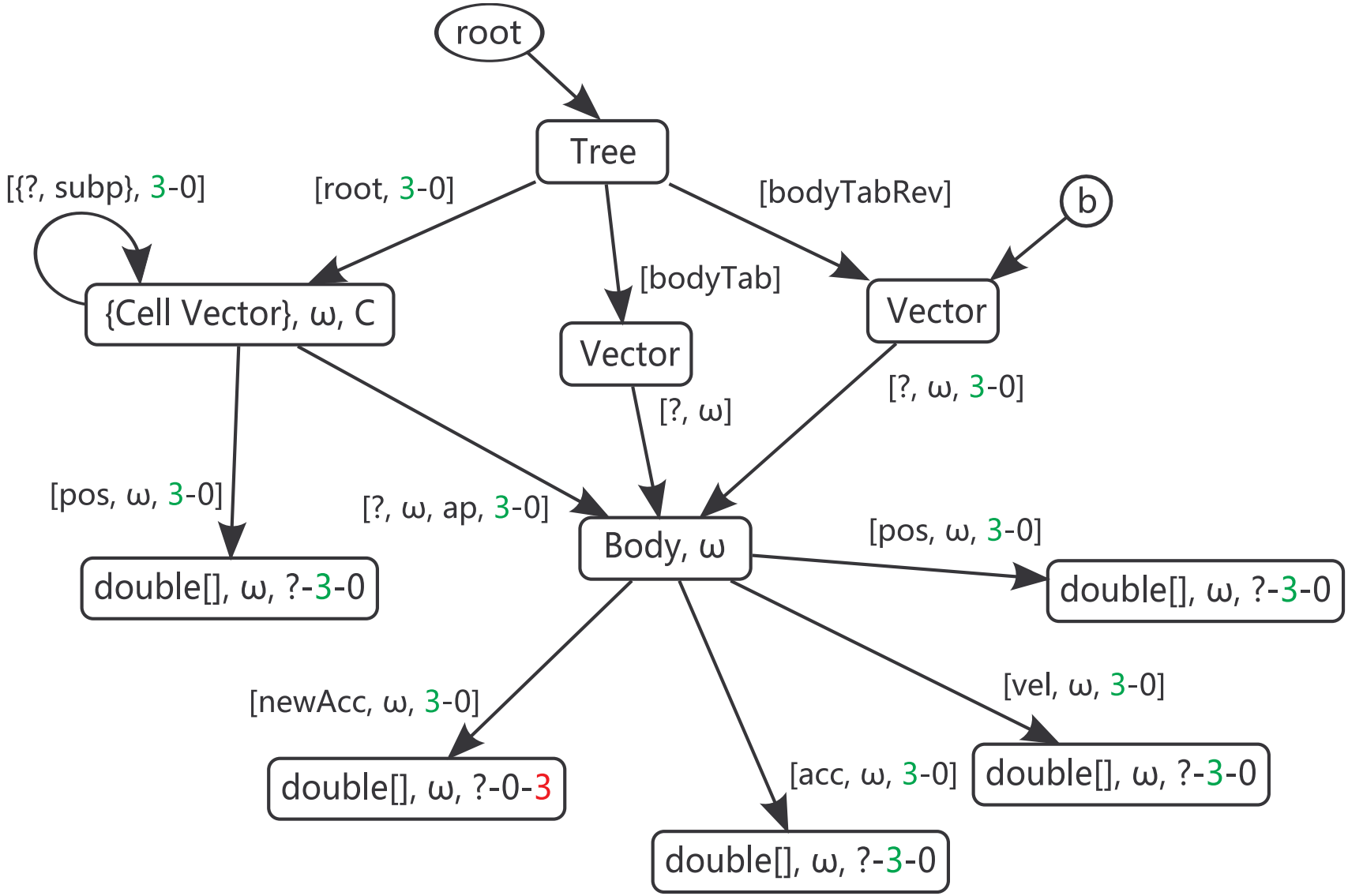
- Can be used for:
  - Parallelizing programs (independence detection).
  - Verifying/debugging user-provided parallel code.
  - Certifying parallel code.



Example: Heap Dependency Analysis (em3d, r-w deps)



Example: Heap Dependency Analysis (bh, r-w deps)

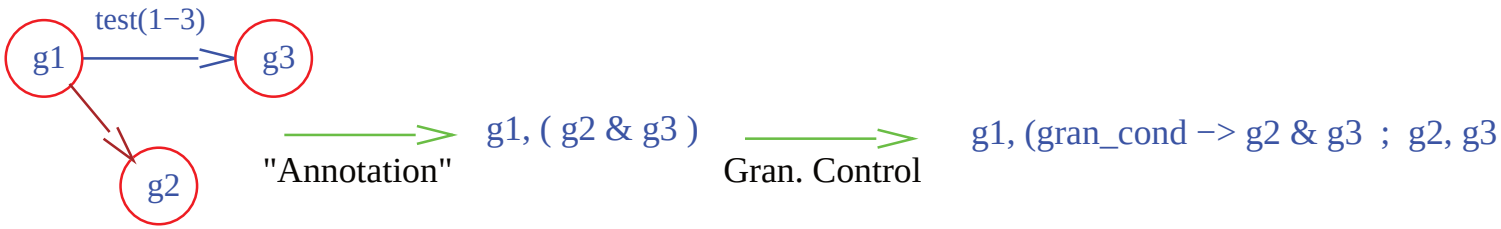


Granularity Control

- Replace parallel with seq. execution based on task size & overheads.
- Cannot be done completely at compile-time: cost often depends on input (hard to approximate precisely even w/abstract interpretation).

```
main :- read(X), read(Z), inc_all(X,Y) & r(Z,M), ...
```

- Our approach:
  - Derive at compile-time cost *functions* (to be evaluated at run-time) that efficiently bound task size (lower, upper *bounds*).
  - Transform programs to carry out run-time granularity control.



- For `inc_all`, (assuming "threshold" is 100 units):

```
main :- read(X), read(Z), ( 2*length(X)+1 > 100 -> inc_all(X,Y) & r(Z,M)
                                                 ;  inc_all(X,Y) , r(Z,M) ),
```

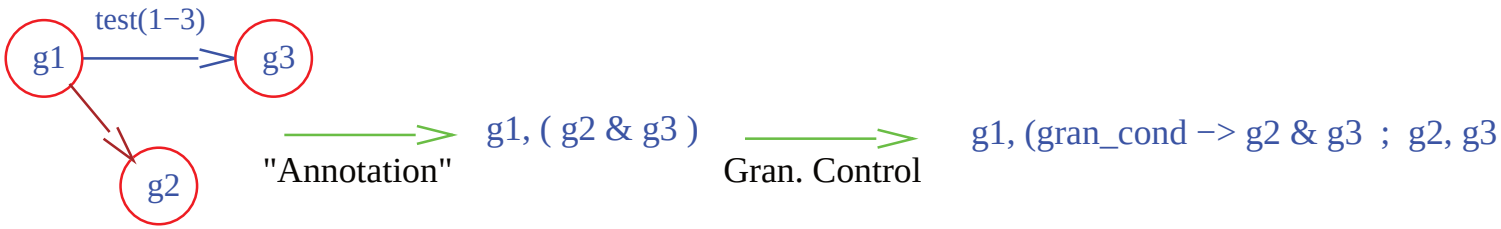
[DLH90, LGHD94, LGHD96, DLGHL94, DLGHL97, MLGCH08]

Granularity Control

- Replace parallel with seq. execution based on task size & overheads.
- Cannot be done completely at compile-time: cost often depends on input (hard to approximate precisely even w/abstract interpretation).

```
main :- read(X), read(Z), inc_all(X,Y) & r(Z,M), ...
```

- Our approach:
  - Derive at compile-time cost *functions* (to be evaluated at run-time) that efficiently bound task size (lower, upper *bounds*).
  - Transform programs to carry out run-time granularity control.



- For `inc_all`, (assuming "threshold" is 100 units):

```
main :- read(X), read(Z), ( 2*length(X)+1 > 100 -> inc_all(X,Y) & r(Z,M)
                                                 ;  inc_all(X,Y) , r(Z,M) ),
```

[DLH90, LGHD94, LGHD96, DLGHL94, DLGHL97, MLGCH08]

Refinements: Granularity Control Optimizations

- Simplification of cost functions:

```
..., ( length(X) > 50 -> inc_all(X,Y) & r(Z,M)
                       ;  inc_all(X,Y) , r(Z,M) ), ...
```

```
..., ( length_gt(LX,50) -> inc_all(X,Y) & r(Z,M)
                         ;  inc_all(X,Y) , r(Z,M) ), ...
```

- Complex thresholds: use also communication cost functions, load, ...

  Example: Assume $CommCost(inc\_all(X)) = 0.1\ (length(X) + length(Y))$

  We know $ub\_length(Y)$ (actually, exact size) $= length(X)$; thus:

$$2\ length(X) + 1 > 0.1\ (length(X) + length(X)) \cong$$
$$2\ length(X) > 0.2\ length(X) \equiv$$

  Guaranteed speedup for any data size! $\Leftarrow$ $2 > 0.2$

- Checking of data sizes can be stopped once under threshold.
- Data size computations can often be done on-the-fly.
- Static task clustering (loop unrolling), static placement, etc.

Refinements: Granularity Control Optimizations

- Simplification of cost functions:

```
..., ( length(X) > 50 -> inc_all(X,Y) & r(Z,M)
                       ;  inc_all(X,Y) , r(Z,M) ), ...

..., ( length_gt(LX,50) -> inc_all(X,Y) & r(Z,M)
                         ;  inc_all(X,Y) , r(Z,M) ), ...
```

- Complex thresholds: use also communication cost functions, load, ...

  Example: Assume $CommCost(inc\_all(X)) = 0.1\ (length(X) + length(Y))$

  We know $ub\_length(Y)$ (actually, exact size) $= length(X)$; thus:

$$2\ length(X) + 1 > 0.1\ (length(X) + length(X)) \cong$$
$$2\ length(X) > 0.2\ length(X) \equiv$$

  Guaranteed speedup for any data size! $\Leftarrow$ $2 > 0.2$

- Checking of data sizes can be stopped once under threshold.
- Data size computations can often be done on-the-fly.
- Static task clustering (loop unrolling), static placement, etc.

Refinements: Granularity Control Optimizations

- Simplification of cost functions:

```
..., ( length(X) > 50 -> inc_all(X,Y) & r(Z,M)
                       ;  inc_all(X,Y) , r(Z,M) ), ...

..., ( length_gt(LX,50) -> inc_all(X,Y) & r(Z,M)
                         ;  inc_all(X,Y) , r(Z,M) ), ...
```

- Complex thresholds: use also communication cost functions, load, ...

  Example: Assume $CommCost(inc\_all(X)) = 0.1\ (length(X) + length(Y))$

  We know $ub\_length(Y)$ (actually, exact size) $= length(X)$; thus:

$$2\ length(X) + 1 > 0.1\ (length(X) + length(X)) \cong$$
$$2\ length(X) > 0.2\ length(X) \equiv$$
$$2 > 0.2 \Leftarrow \text{Guaranteed speedup for any data size!}$$

- Checking of data sizes can be stopped once under threshold.
- Data size computations can often be done on-the-fly.
- Static task clustering (loop unrolling), static placement, etc.

Refinements: Granularity Control Optimizations

- Simplification of cost functions:

```
..., ( length(X) > 50 -> inc_all(X,Y) & r(Z,M)
                       ;  inc_all(X,Y) , r(Z,M) ), ...

..., ( length_gt(LX,50) -> inc_all(X,Y) & r(Z,M)
                         ;  inc_all(X,Y) , r(Z,M) ), ...
```

- Complex thresholds: use also communication cost functions, load, ...

  Example: Assume $CommCost(inc\_all(X)) = 0.1\ (length(X) + length(Y))$

  We know $ub\_length(Y)$ (actually, exact size) $= length(X)$; thus:

$$2\ length(X) + 1 > 0.1\ (length(X) + length(X)) \cong$$
$$2\ length(X) > 0.2\ length(X) \equiv$$

  Guaranteed speedup for any data size! $\Leftarrow$ $2 > 0.2$

- Checking of data sizes can be stopped once under threshold.
- Data size computations can often be done on-the-fly.
- Static task clustering (loop unrolling), static placement, etc.

Refinements: Granularity Control Optimizations

- Simplification of cost functions:

```
..., ( length(X) > 50 -> inc_all(X,Y) & r(Z,M)
                       ;  inc_all(X,Y) , r(Z,M) ), ...

..., ( length_gt(LX,50) -> inc_all(X,Y) & r(Z,M)
                         ;  inc_all(X,Y) , r(Z,M) ), ...
```

- Complex thresholds: use also communication cost functions, load, ...

  Example: Assume $CommCost(inc\_all(X)) = 0.1\ (length(X) + length(Y))$

  We know $ub\_length(Y)$ (actually, exact size) $= length(X)$; thus:

$$2\ length(X) + 1 > 0.1\ (length(X) + length(X)) \cong$$
$$2\ length(X) > 0.2\ length(X) \equiv$$

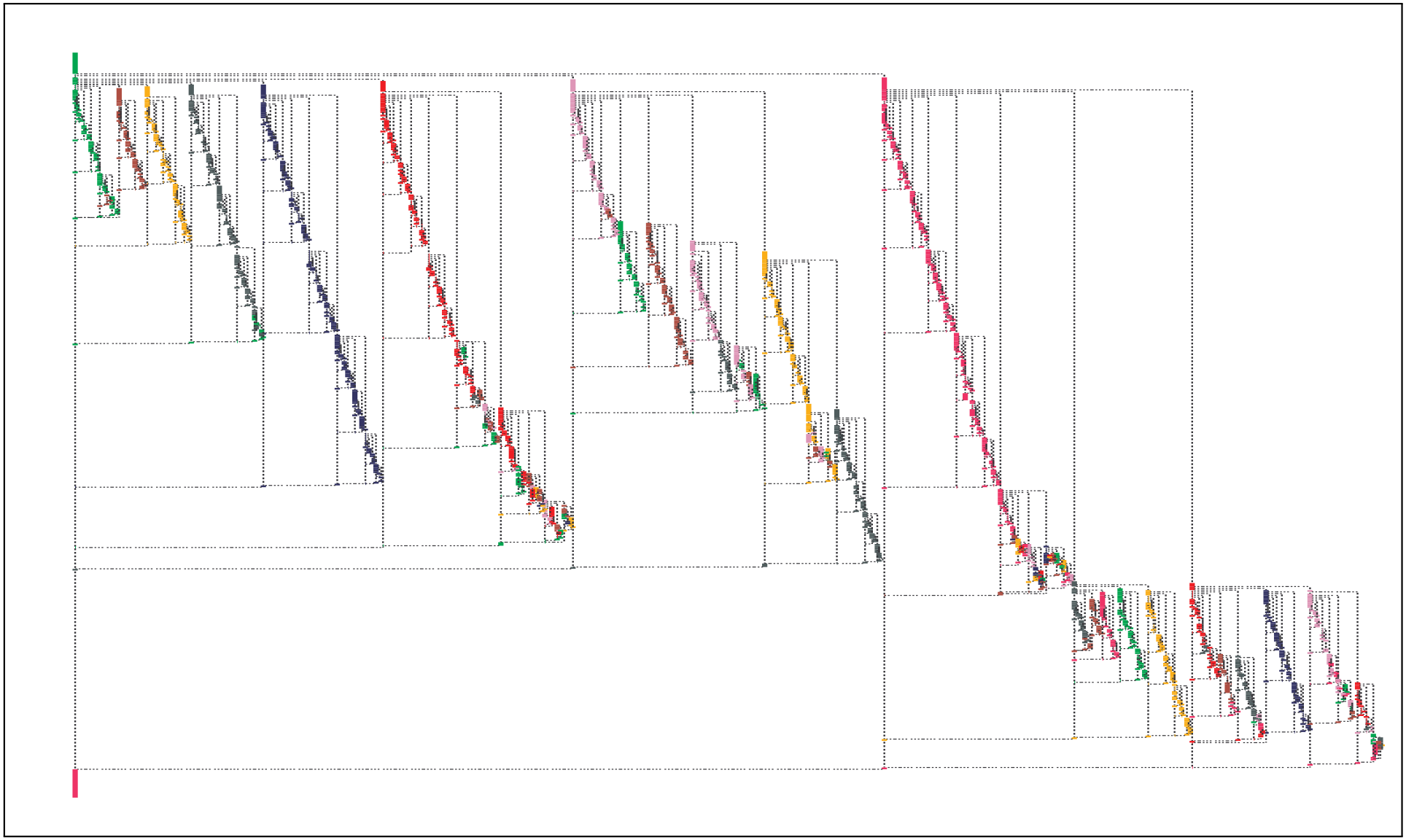
  Guaranteed speedup for any data size! $\Leftarrow$ $2 > 0.2$

- Checking of data sizes can be stopped once under threshold.
- Data size computations can often be done on-the-fly.
- Static task clustering (loop unrolling), static placement, etc.

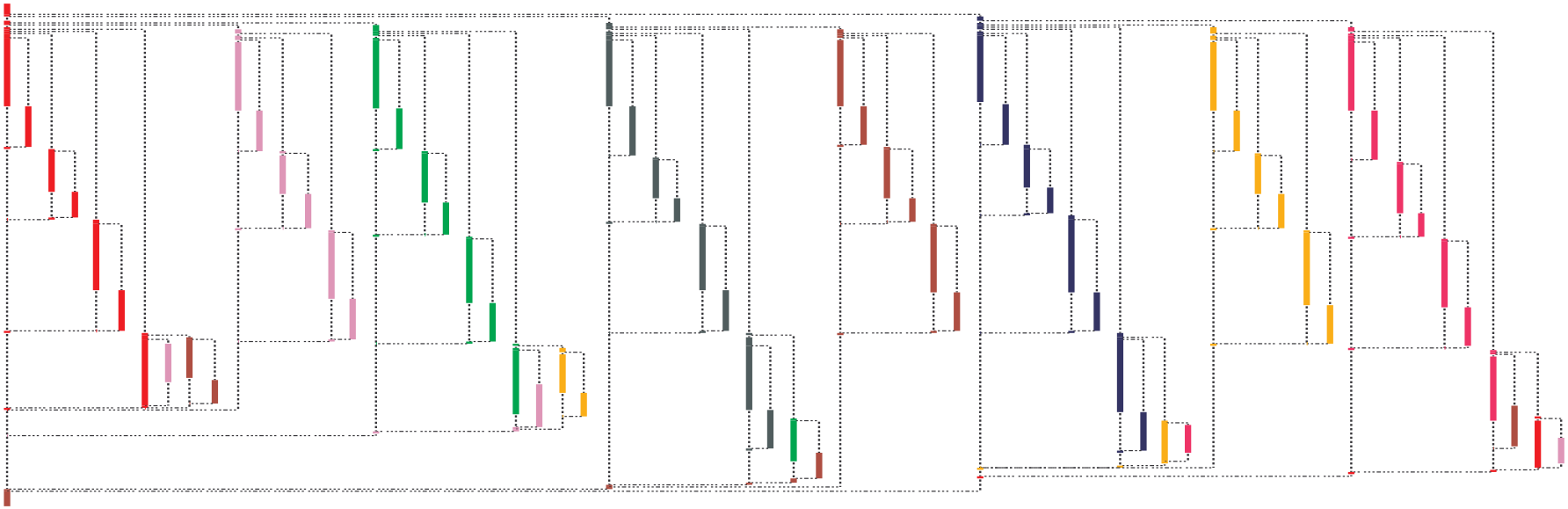
Granularity Control System Output Example

```
g_qsort([], []).
g_qsort([First|L1], L2) :-
  partition3o4o(First, L1, Ls, Lg, Size_Ls, Size_Lg),
  Size_Ls > 20 -> (Size_Lg > 20 -> g_qsort(Ls, Ls2) & g_qsort(Lg, Lg2)
                                ;  g_qsort(Ls, Ls2) , s_qsort(Lg, Lg2))
                ;  (Size_Lg > 20 -> s_qsort(Ls, Ls2) , g_qsort(Lg, Lg2)
                                ;  s_qsort(Ls, Ls2) , s_qsort(Lg, Lg2))),
  append(Ls2, [First|Lg2], L2).

partition3o4o(F, [], [], [], 0, 0).
partition3o4o(F, [X|Y], [X|Y1], Y2, SL, SG) :-
  X =< F, partition3o4o(F, Y, Y1, Y2, SL1, SG), SL is SL1 + 1.
partition3o4o(F, [X|Y], Y1, [X|Y2], SL, SG) :-
   X > F, partition3o4o(F, Y, Y1, Y2, SL, SG1), SG is SG1 + 1.
```

8 processors



8 processors, with granularity control (same scale)



Summarized System Timeline

- The original problem: auto-parallelization (1983).
- MA3 System - 1988 - Memo tables, practicality.
- 1989 - PLAI framework/fixpoint. Set sharing, side-effects.
- 1990 - First cost analysis (for task granularity), upper bounds.
- 1991 - Sharing+Freenes, def (dependencies).
- Early 90's - Practical auto-parallelization.
- 90's - Incrementality, modularity, extension to constraint programming, concurrency (dynamic scheduling), domain combinations.
- Mid 90's, 2006 - Combination with partial evaluation, lower bounds.
- Mid-Late 90's - CiaoPP: Integrated verification, debugging, optimization.
- Late 90's - Non-failure (no exceptions), determinacy,
- 2001-05 - Modularity, diagnosis.
- 2001-... Verification of cost, resources, ...
- 2002-... New shape/type domains, widenings.
- 2003-... Abstraction carrying code, reduced certificates, ...
- 2003-... User-defined resources. Time, energy, ...
- 2005-... *Multi-language support*, Java, C# (shapes, resources, ...).

Final Remarks

- Instead of full specifications a priori (waterfall):
  - Develop program and specifications gradually, not necessarily in sync.
  - Both can be incomplete (including types!).
    - Temporarily use spec (including tests) as implementation.
  - Go from types, to more complex assertions, to full specifications.
    - Can incrementally strengthen them to defs of full functional correctness.

  Safe approximations / abstractions everywhere –essential role!
- View debugging, verification, certification, testing, optimization *in an integrated way:* strong synergy.
  - E.g., tests as part of the specification.
- FMs as integral part of the development cycle ("programmer's tools").
  - Need to integrate in standard tool chain.
- Assertion language design is important: many roles, used throughout.
- Assertions, properties in source language; "seamless integration."
- Multi-language analysis through IR; programs in several languages.

Final Remarks

- Instead of full specifications a priori (waterfall):
  - Develop program and specifications gradually, not necessarily in sync.
  - Both can be incomplete (including types!).
    - Temporarily use spec (including tests) as implementation.
  - Go from types, to more complex assertions, to full specifications.
    - Can incrementally strengthen them to defs of full functional correctness.

  Safe approximations / abstractions everywhere –essential role!
- View debugging, verification, certification, testing, optimization *in an integrated way:* strong synergy.
  - E.g., tests as part of the specification.
- FMs as integral part of the development cycle ("programmer's tools").
  - Need to integrate in standard tool chain.
- Assertion language design is important: many roles, used throughout.
- Assertions, properties in source language; "seamless integration."
- Multi-language analysis through IR; programs in several languages.

Final Remarks

- Instead of full specifications a priori (waterfall):
  - Develop program and specifications gradually, not necessarily in sync.
  - Both can be incomplete (including types!).
    - Temporarily use spec (including tests) as implementation.
  - Go from types, to more complex assertions, to full specifications.
    - Can incrementally strengthen them to defs of full functional correctness.

  Safe approximations / abstractions everywhere –essential role!
- View debugging, verification, certification, testing, optimization *in an integrated way:* strong synergy.
  - E.g., tests as part of the specification.
- FMs as integral part of the development cycle ("programmer's tools").
  - Need to integrate in standard tool chain.
- Assertion language design is important: many roles, used throughout.
- Assertions, properties in source language; "seamless integration."
- Multi-language analysis through IR; programs in several languages.

Final Remarks

- Instead of full specifications a priori (waterfall):
  - Develop program and specifications gradually, not necessarily in sync.
  - Both can be incomplete (including types!).
    - Temporarily use spec (including tests) as implementation.
  - Go from types, to more complex assertions, to full specifications.
    - Can incrementally strengthen them to defs of full functional correctness.

  Safe approximations / abstractions everywhere –essential role!
- View debugging, verification, certification, testing, optimization *in an integrated way:* strong synergy.
  - E.g., tests as part of the specification.
- FMs as integral part of the development cycle ("programmer's tools").
  - Need to integrate in standard tool chain.
- Assertion language design is important: many roles, used throughout.
- Assertions, properties in source language; "seamless integration."
- Multi-language analysis through IR; programs in several languages.

Final Remarks

- Some Plans for CiaoPP/Ciao:
  - Continue multi-language analysis: programs in several languages.
  - Increase scalability by exploiting further modularity and incrementality of underlying technology.
  - Package parts as reusable components (perhaps part of a more general project?).
  - Applications in security (including, e.g., timing attacks).
  - Continue with language design (Ciao).
  - Continue exploring synergies. E.g., combine with theorem proving and/or model checking?

Some Members of The Ciao Forge

- CiaoPP/Ciao is really a widely distributed collaborative effort:
  - Directly within the CLIP Group (UPM and IMDEA Software):
    M. Hermenegildo, K. Muthukumar, M. García de la Banda, F. Bueno, G. Puebla, M. Carro, D. Cabeza, P. López-G., R. Haemmerlé, J. Morales, E. Mera, J. Navas, M. Méndez, A. Casas, J. Correas, D. Trallero, C. Ochoa, P. Chico, M.T. Trigo, P. Pietrzak, C. Vaucheret, E. Albert, P. Arenas, S. Genaim, ...
  - 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, ...

References – Overall Model

[BDD+97] F. Bueno, P. Deransart, W. Drabent, G. Ferrand, M. Hermenegildo, J. Maluszynski, and G. Puebla.
On the Role of Semantic Approximations in Validation and Diagnosis of Constraint Logic Programs.
In *Proc. of the 3rd. Int'l Workshop on Automated Debugging–AADEBUG'97*, pages 155–170, Linköping, Sweden, May 1997. U. of Linköping Press.

[HPB99] M. Hermenegildo, G. Puebla, and F. Bueno.
Using Global Analysis, Partial Specifications, and an Extensible Assertion Language for Program Validation and Debugging.
In K. R. Apt, V. Marek, M. Truszczynski, and D. S. Warren, editors, *The Logic Programming Paradigm: a 25–Year Perspective*, pages 161–192. Springer-Verlag, July 1999.

[PBH00c] G. Puebla, F. Bueno, and M. Hermenegildo.
Combined Static and Dynamic Assertion-Based Debugging of Constraint Logic Programs.
In *Logic-based Program Synthesis and Transformation (LOPSTR'99)*, number 1817 in LNCS, pages 273–292. Springer-Verlag, March 2000.

[PBH00a] G. Puebla, F. Bueno, and M. Hermenegildo.
A Generic Preprocessor for Program Validation and Debugging.
In P. Deransart, M. Hermenegildo, and J. Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming*, number 1870 in LNCS, pages 63–107. Springer-Verlag, September 2000.

[HPBLG03] M. Hermenegildo, G. Puebla, F. Bueno, and P. López-García.
Program Development Using Abstract Interpretation (and The Ciao System Preprocessor).
In *10th International Static Analysis Symposium (SAS'03)*, number 2694 in LNCS, pages 127–152. Springer-Verlag, June 2003.

[MLGH09] E. Mera, P. López-García, and M. Hermenegildo.
Integrating Software Testing and Run-Time Checking in an Assertion Verification Framework.
In *25th International Conference on Logic Programming (ICLP'09)*, number 5649 in LNCS, pages 281–295. Springer-Verlag, July 2009.

References – Assertion Language

[PBH97] G. Puebla, F. Bueno, and M. Hermenegildo.
An Assertion Language for Debugging of Constraint Logic Programs.
In *Proceedings of the ILPS'97 Workshop on Tools and Environments for (Constraint) Logic Programming*, October 1997.
Available from ftp://clip.dia.fi.upm.es/pub/papers/assert_lang_tr_discipldeliv.ps.gz as technical report CLIP2/97.1.

[PBH00b] G. Puebla, F. Bueno, and M. Hermenegildo.
An Assertion Language for Constraint Logic Programs.
In P. Deransart, M. Hermenegildo, and J. Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming*, number 1870 in LNCS, pages 23–61. Springer-Verlag, September 2000.

[MLGH09] E. Mera, P. López-García, and M. Hermenegildo.
Integrating Software Testing and Run-Time Checking in an Assertion Verification Framework.
In *25th International Conference on Logic Programming (ICLP'09)*, number 5649 in LNCS, pages 281–295. Springer-Verlag, July 2009.

References – Intermediate Representation

[MLNH07] M. Méndez-Lojo, J. Navas, and M. Hermenegildo.
A Flexible (C)LP-Based Approach to the Analysis of Object-Oriented Programs.
In *17th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR 2007)*, number 4915 in LNCS, pages 154–168. Springer-Verlag, August 2007.

References – Abstraction Carrying Code

[AAPH06] E. Albert, P. Arenas, G. Puebla, and M. Hermenegildo.
Reduced Certificates for Abstraction-Carrying Code.
In *22nd International Conference on Logic Programming (ICLP 2006)*, number 4079 in LNCS, pages 163–178. Springer-Verlag, August 2006.

[APH05] E. Albert, G. Puebla, and M. Hermenegildo.
Abstraction-Carrying Code.
In *Proc. of LPAR'04*, volume 3452 of *LNAI*. Springer, 2005.

[HALGP05] M. Hermenegildo, E. Albert, P. López-García, and G. Puebla.
Abstraction Carrying Code and Resource-Awareness.
In *PPDP*. ACM Press, 2005.

References – Fixpoint-based Analyzers (Abstract Interpreters)

[MH92] K. Muthukumar and M. Hermenegildo.
Compile-time Derivation of Variable Dependency Using Abstract Interpretation.
Journal of Logic Programming, 13(2/3):315–347, July 1992.

[BGH99] F. Bueno, M. García de la Banda, and M. Hermenegildo.
Effectiveness of Abstract Interpretation in Automatic Parallelization: A Case Study in Logic Programming.
ACM Transactions on Programming Languages and Systems, 21(2):189–238, March 1999.

[PH96] G. Puebla and M. Hermenegildo.
Optimized Algorithms for the Incremental Analysis of Logic Programs.
In *International Static Analysis Symposium (SAS 1996)*, number 1145 in LNCS, pages 270–284. Springer-Verlag, September 1996.

[HPMS00] M. Hermenegildo, G. Puebla, K. Marriott, and P. Stuckey.
Incremental Analysis of Constraint Logic Programs.
ACM Transactions on Programming Languages and Systems, 22(2):187–223, March 2000.

[NMLH07] J. Navas, M. Méndez-Lojo, and M. Hermenegildo.
An Efficient, Context and Path Sensitive Analysis Framework for Java Programs.
In *9th Workshop on Formal Techniques for Java-like Programs FTfJP 2007*, July 2007.

References – Modular Analysis, Analysis of Concurrency

[MGH94] K. Marriott, M. García de la Banda, and M. Hermenegildo.
Analyzing Logic Programs with Dynamic Scheduling.
In *20th. Annual ACM Conf. on Principles of Programming Languages*, pages 240–254. ACM, January 1994.

[BCHP96] F. Bueno, D. Cabeza, M. Hermenegildo, and G. Puebla.
Global Analysis of Standard Prolog Programs.
In *European Symposium on Programming*, number 1058 in LNCS, pages 108–124, Sweden, April 1996. Springer-Verlag.

[PH00] G. Puebla and M. Hermenegildo.
Some Issues in Analysis and Specialization of Modular Ciao-Prolog Programs.
In *Special Issue on Optimization and Implementation of Declarative Programming Languages*, volume 30 of *Electronic Notes in Theoretical Computer Science*. Elsevier - North Holland, March 2000.

[BdlBH+01] F. Bueno, M. García de la Banda, M. Hermenegildo, K. Marriott, G. Puebla, and P. Stuckey.
A Model for Inter-module Analysis and Optimizing Compilation.
In *Logic-based Program Synthesis and Transformation*, number 2042 in LNCS, pages 86–102. Springer-Verlag, March 2001.

[PCPH06] P. Pietrzak, J. Correas, G. Puebla, and M. Hermenegildo.
Context-Sensitive Multivariant Assertion Checking in Modular Programs.
In *LPAR'06*, number 4246 in LNCS, pages 392–406. Springer-Verlag, November 2006.

[PCPH08] P. Pietrzak, J. Correas, G. Puebla, and M. Hermenegildo.
A Practical Type Analysis for Verification of Modular Prolog Programs.
In *PEPM'08*, pages 61–70. ACM Press, January 2008.

References – Domains: Sharing/Aliasing

[MH89] K. Muthukumar and M. Hermenegildo.
Determination of Variable Dependence Information at Compile-Time Through Abstract Interpretation.
In *1989 North American Conf. on Logic Programming*, pages 166–189. MIT Press, October 1989.

[MH91] K. Muthukumar and M. Hermenegildo.
Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation.
In *International Conference on Logic Programming (ICLP 1991)*, pages 49–63. MIT Press, June 1991.

[NBH06] J. Navas, F. Bueno, and M. Hermenegildo.
Efficient top-down set-sharing analysis using cliques.
In *Eight International Symposium on Practical Aspects of Declarative Languages*, number 2819 in LNCS, pages 183–198. Springer-Verlag, January 2006.

[MLH08] M. Méndez-Lojo and M. Hermenegildo.
Precise Set Sharing Analysis for Java-style Programs.
In *9th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'08)*, number 4905 in LNCS, pages 172–187. Springer-Verlag, January 2008.

[MMLH+08] M. Marron, M. Méndez-Lojo, M. Hermenegildo, D. Stefanovic, and D. Kapur.
Sharing Analysis of Arrays, Collections, and Recursive Structures.
In *ACM WS on Program Analysis for SW Tools and Engineering (PASTE'08)*. ACM, November 2008.

[MKSH08] M. Marron, D. Kapur, D. Stefanovic, and M. Hermenegildo.
Identification of Heap-Carried Data Dependence Via Explicit Store Heap Models.
In *21st Int'l. WS on Languages and Compilers for Parallel Computing (LCPC'08)*, LNCS. Springer-Verlag, August 2008.

[MLLH08] M. Méndez-Lojo, O. Lhoták, and M. Hermenegildo.
Efficient Set Sharing using ZBDDs.
In *21st Int'l. WS on Languages and Compilers for Parallel Computing (LCPC'08)*, LNCS. Springer-Verlag, August 2008.

[MKH09] M. Marron, D. Kapur, and M. Hermenegildo.
Identification of Logically Related Heap Regions.
In *ISMM'09: Proceedings of the 8th international symposium on Memory management*, New York, NY, USA, June 2009. ACM Press.

References – Domains: Shape/Type Analysis

[VB02] C. Vaucheret and F. Bueno.
More Precise yet Efficient Type Inference for Logic Programs.
In *International Static Analysis Symposium*, volume 2477 of *Lecture Notes in Computer Science*, pages 102–116. Springer-Verlag, September 2002.

[MSHK07] M. Marron, D. Stefanovic, M. Hermenegildo, and D. Kapur.
Heap Analysis in the Presence of Collection Libraries.
In *ACM WS on Program Analysis for Software Tools and Engineering (PASTE'07)*. ACM, June 2007.

[MHKS08] M. Marron, M. Hermenegildo, D. Kapur, and D. Stefanovic.
Efficient context-sensitive shape analysis with graph-based heap models.
In Laurie Hendren, editor, *International Conference on Compiler Construction (CC 2008)*, Lecture Notes in Computer Science. Springer, April 2008.

References – Domains: Non-failure, Determinacy

[DLGH97] S.K. Debray, P. López-García, and M. Hermenegildo.
Non-Failure Analysis for Logic Programs.
In *1997 International Conference on Logic Programming*, pages 48–62, Cambridge, MA, June 1997. MIT Press, Cambridge, MA.

[BLGH04] F. Bueno, P. López-García, and M. Hermenegildo.
Multivariant Non-Failure Analysis via Standard Abstract Interpretation.
In *7th International Symposium on Functional and Logic Programming (FLOPS 2004)*, number 2998 in LNCS, pages 100–116, Heidelberg, Germany, April 2004. Springer-Verlag.

[LGBH05] P. López-García, F. Bueno, and M. Hermenegildo.
Determinacy Analysis for Logic Programs Using Mode and Type Information.
In *Proceedings of the 14th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'04)*, number 3573 in LNCS, pages 19–35. Springer-Verlag, August 2005.

[LGBH10] P. López-García, F. Bueno, and M. Hermenegildo.
Automatic Inference of Determinacy and Mutual Exclusion for Logic Programs Using Mode and Type Information.
New Generation Computing, 28(2):117–206, 2010.

References – Analysis and Verification of Resources

[DLH90] S. K. Debray, N.-W. Lin, and M. Hermenegildo.
Task Granularity Analysis in Logic Programs.
In *Proc. of the 1990 ACM Conf. on Programming Language Design and Implementation*, pages 174–188. ACM Press, June 1990.

[LGHD94] P. López-García, M. Hermenegildo, and S.K. Debray.
Towards Granularity Based Control of Parallelism in Logic Programs.
In Hoon Hong, editor, *Proc. of First International Symposium on Parallel Symbolic Computation, PASCO'94*, pages 133–144. World Scientific, September 1994.

[LGHD96] P. López-García, M. Hermenegildo, and S. K. Debray.
A Methodology for Granularity Based Control of Parallelism in Logic Programs.
Journal of Symbolic Computation, Special Issue on Parallel Symbolic Computation, 21(4–6):715–734, 1996.

[DLGHL94] S.K. Debray, P. López-García, M. Hermenegildo, and N.-W. Lin.
Estimating the Computational Cost of Logic Programs.
In *Static Analysis Symposium, SAS'94*, number 864 in LNCS, pages 255–265, Namur, Belgium, September 1994. Springer-Verlag.

[DLGHL97] S. K. Debray, P. López-García, M. Hermenegildo, and N.-W. Lin.
Lower Bound Cost Estimation for Logic Programs.
In *1997 International Logic Programming Symposium*, pages 291–305. MIT Press, Cambridge, MA, October 1997.

[NMLGH07] J. Navas, E. Mera, P. López-García, and M. Hermenegildo.
User-Definable Resource Bounds Analysis for Logic Programs.
In *23rd International Conference on Logic Programming (ICLP'07)*, volume 4670 of *Lecture Notes in Computer Science*. Springer, 2007.

[MLGCH08] E. Mera, P. López-García, M. Carro, and M. Hermenegildo.
Towards Execution Time Estimation in Abstract Machine-Based Languages.
In *10th Int'l. ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP'08)*, pages 174–184. ACM Press, July 2008.

[NMLH08] J. Navas, M. Méndez-Lojo, and M. Hermenegildo.
Safe Upper-bounds Inference of Energy Consumption for Java Bytecode Applications.
In *The Sixth NASA Langley Formal Methods Workshop (LFM 08)*, April 2008.
Extended Abstract.

[NMLH09] J. Navas, M. Méndez-Lojo, and M. Hermenegildo.
User-Definable Resource Usage Bounds Analysis for Java Bytecode.
In *Proceedings of the Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE'09)*, volume 253 of *Electronic Notes in Theoretical Computer Science*, pages 6–86. Elsevier - North Holland, March 2009.

[LGDB10] P. López-García, L. Darmawan, and F. Bueno.
A Framework for Verification and Debugging of Resource Usage Properties.
In M. Hermenegildo and T. Schaub, editors, *Technical Communications of the 26th Int'l. Conference on Logic Programming (ICLP'10)*, volume 7 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 104–113, Dagstuhl, Germany, July 2010. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

References – Automatic Parallelization, (Abstract) Partial Evaluation, Other Optimizations

[GH91] F. Giannotti and M. Hermenegildo.
A Technique for Recursive Invariance Detection and Selective Program Specialization.
In *Proc. 3rd. Int'l Symposium on Programming Language Implementation and Logic Programming*, number 528 in LNCS, pages 323–335. Springer-Verlag, August 1991.

[PH97] G. Puebla and M. Hermenegildo.
Abstract Specialization and its Application to Program Parallelization.
In J. Gallagher, editor, *Logic Program Synthesis and Transformation*, number 1207 in LNCS, pages 169–186. Springer-Verlag, 1997.

[MBdlBH99] K. Muthukumar, F. Bueno, M. García de la Banda, and M. Hermenegildo.
Automatic Compile-time Parallelization of Logic Programs for Restricted, Goal-level, Independent And-parallelism.
Journal of Logic Programming, 38(2):165–218, February 1999.

[PH99] G. Puebla and M. Hermenegildo.
Abstract Multiple Specialization and its Application to Program Parallelization.
J. of Logic Programming. Special Issue on Synthesis, Transformation and Analysis of Logic Programs, 41(2&3):279–316, November 1999.

[PHG99] G. Puebla, M. Hermenegildo, and J. Gallagher.
An Integration of Partial Evaluation in a Generic Abstract Interpretation Framework.
In O Danvy, editor, *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'99)*, number NS-99-1 in BRISC Series, pages 75–85. University of Aarhus, Denmark, January 1999.

[PH03] G. Puebla and M. Hermenegildo.
Abstract Specialization and its Applications.
In *ACM Partial Evaluation and Semantics based Program Manipulation (PEPM'03)*, pages 29–43. ACM Press, June 2003.
Invited talk.

[PAH06] G. Puebla, E. Albert, and M. Hermenegildo.
Abstract Interpretation with Specialized Definitions.
In *SAS'06*, number 4134 in LNCS, pages 107–126. Springer-Verlag, 2006.

[CCH08] A. Casas, M. Carro, and M. Hermenegildo.
A High-Level Implementation of Non-Deterministic, Unrestricted, Independent And-Parallelism.
In M. García de la Banda and E. Pontelli, editors, *24th International Conference on Logic Programming (ICLP'08)*, volume 5366 of *LNCS*, pages 651–666. Springer-Verlag, December 2008.

[MKSH08] M. Marron, D. Kapur, D. Stefanovic, and M. Hermenegildo.
Identification of Heap-Carried Data Dependence Via Explicit Store Heap Models.
In *21st Int'l. WS on Languages and Compilers for Parallel Computing (LCPC'08)*, LNCS. Springer-Verlag, August 2008.

[MCH04] J. Morales, M. Carro, and M. Hermenegildo.
Improving the Compilation of Prolog to C Using Moded Types and Determinism Information.
In *PADL'04*, number 3057 in LNCS, pages 86–103. Springer-Verlag, June 2004.

[CMM+06] M. Carro, J. Morales, H.L. Muller, G. Puebla, and M. Hermenegildo.
High-Level Languages for Small Devices: A Case Study.
In Krisztian Flautner and Taewhan Kim, editors, *Compilers, Architecture, and Synthesis for Embedded Systems*, pages 271–281. ACM Press / Sheridan, October 2006.