Analysis and Verification "of and with" Horn Clauses (using the Ciao system)

Manuel Hermenegildo [1,2] M. Carro [1,2] P. López-García [3,1] U. Liqat [1]
J. Morales [1] P. Chico [4] R. Haemmerlé [2] A. Serrano [1]

[1] IMDEA Software Institute
[2] Technical University of Madrid (UPM)
[3] Spanish Research Council (CSIC)
[4] Elasticbox

COST Rich Model Toolkit Workshop, Malta, June 16-17, 2013

Outline

Part I The Ciao approach to Analysis and verification
of Constraint Logic Programs

- The programming language
- The analysis, verification, and testing model

Part II The Ciao approach to Analysis and verification
of other paradigms
using Constraint Logic Programs as IR

- CLP (Horn Clauses) as intermediate representation
- User-defined resource analysis/verif. of Java bytecode
- Energy analysis/verification of (Xmos) C programs

Outline

Part I The Ciao approach to Analysis and verification
of Constraint Logic Programs

- The programming language
- The analysis, verification, and testing model

Part II The Ciao approach to Analysis and verification
of other paradigms
using Constraint Logic Programs as IR

- CLP (Horn Clauses) as intermediate representation
- User-defined resource analysis/verif. of Java bytecode
- Energy analysis/verification of (Xmos) C programs

Logic and constraint programming: Mid-90's:

- Prolog/CLPs (dynamic), Mercury (static), Ciao (combination).
- *Static analysis* (abstract interpretation) maturing (aliasing, modes, data sizes, execution cost, ... scalability, incrementality, ...)

The Ciao approach [CP'94,AADEBUG'97,ICLP'99,...]

- Start from a small, but very *extensible* (LP-based) kernel – a language-building language.
- Build gradually extensions in layers on top of it.
- Incorporating the *most useful features* from different prog. paradigms.
- Offer the *best of the dynamic and static* language approaches.
  - Provide the flexibility of dynamic languages,
    - Dynamic typing, dynamic load, dynamic program modification, meta-programming, top level, call (eval), scripts, ...
  - But with *guaranteed safety and efficiency.*
    - *Assertion checking*, modules, itf files, separate/incr. compilation, small executables, embeddability, high-performance, ...
- Support the programmer with a *great environment*.

Ciao Enablers

- Module system design:
  - Allows separating dynamic and static code.
  - Allows global analysis, separate/incremental compilation.
- Syntactic and semantic extension mechanism (*packages*):
  - All language features are in libraries (loaded, combined per module):
    - Predicates, functions, higher order, constraints, objects, ...
    - Tabling, other search rules, ASP, ... concurrency, parallelism.
    - Full ISO-Prolog support –also via a library.
- The Ciao assertions model
  - Optional assertions, expressing rich (possibly undecidable) properties.
  - Integrated verification/certification, testing, diagnosis (in comp. loop).
  - Use throughout of *safe* approx. (abstract interpretation), "best effort."

- Compile-time and run-time technology:
  - Analysis, partial evaluation, profiling, ...
  - Several back ends (*including Javascript*)
  - Also bytecode (abstract machine written in Ciao dialect, specializable)

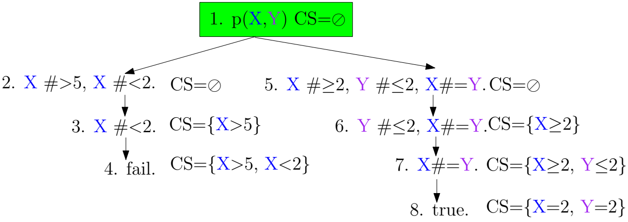
High performance through optimization, not language restriction.

Extension: Constraint Logic Programming

- Natural extension of LP: very general relations between variables allowed (beyond Herbrand term equality).
- Execution inserts new constraints in the constraint store (CS).
- Constraint solver checks consistency of CS.

Example

```
p(X,Y) :-
   X #> 5,
   X #< 2.
p(X,Y) :-
   X #>= 2,
   Y #=< 2,
   X #= Y.

?- p(X,Y).
X = 2, Y = 2
```



Extension: Tabling (OLDT resolution)

- Properties:
  - Conservative extension of Prolog/SLD.
  - Avoids recomputations.
  - Better termination properties; easier to reason about termination.
    - Ensures termination for "bounded term size" programs.
    - In other cases, less dependent on clause / subgoal order.
- Applications:
  - Deductive databases.
  - Natural language (left recursive grammars).
  - Fixpoint: program analysis, reachability analysis...
  - Well Founded Semantics:
    - A predicate can be defined based on its negation.
    - Semantic web reasoning.
  - ...

CLP+Tabling

- Early work:
  - Theoretical; deductive databases, *bottom-up* deduction.
- Goal-directed, top-down poses interesting questions.
  - Existing approaches in LP: XSB, TCHR, **Ciao TCLP**.
  - Still evolving.
- Some issues:
  - Checking applicability of calls and previous solutions: entailment (vs., e.g., call variant or call abstraction)

Goal	Answers
	$X > 3 \wedge Y = 1$
`{X > 3} p(X, Y)`	$X > 3 \wedge Y = 2$
	$X > 5 \wedge Y = 3$

What can we say about `{X > 4} p(X, Y)`?

  - Answers to new (subsumed) calls: conj. of input + answer constraints.

Goal	Answers
	$X > 4 \wedge X > 3 \wedge Y = 1 \equiv X > 4 \wedge Y = 1$
`{X > 4} p(X, Y)`	$X > 4 \wedge X > 3 \wedge Y = 2 \equiv X > 4 \wedge Y = 2$
	$X > 4 \wedge X > 5 \wedge Y = 3 \equiv X > 5 \wedge Y = 3$

  - Non subsumed calls: cannot use stored answer constraint safely.
  - Useful to *project* constraint store on call variables.

Tabled CLP applications

Some Experiments with Timed Automata

- UPPAAL is a fast tool built specifically for TA verification:
  - Developed since 1999.
- Ciao is a general-purpose, multi-paradigm language.

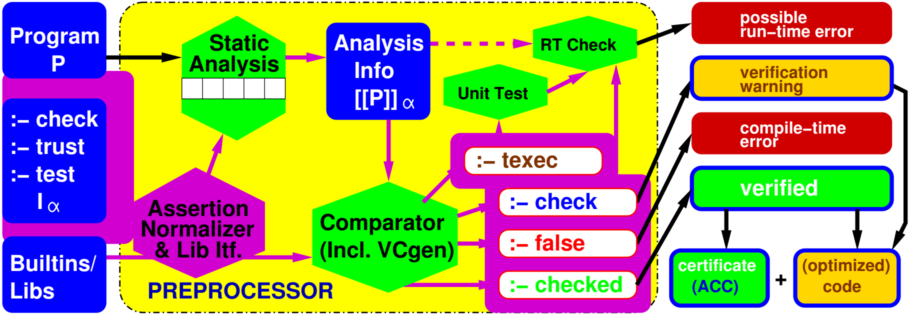
	Ciao TCLP	**UPPAAL**
Fisher 2	0	0
Fisher 3	12	1
Fisher 4	270	44
Fisher 5	10 576	4 514

- Tried to select comparable UPPAAL and Ciao options.
- Additionally: in Ciao, full programming power.

Demo: properties, types, predicates, functions, higher order,

constraints, breadth-first search, tabling, ...

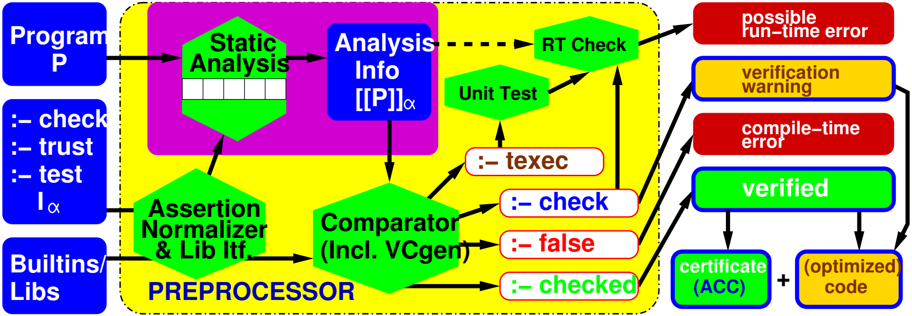
The Assertion Language



- Assertions optional, can be added at any time. Provide partial spec.
- *Sets* of pre/post/global triples (+ "status" field, documentation, ...).
- Used everywhere, for many purposes (incl. doc gen., foreign itf).
- System makes it worthwhile for the programmer to include them.
- Part of the programming language and "runnable."

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

The Analyses (will return to them)

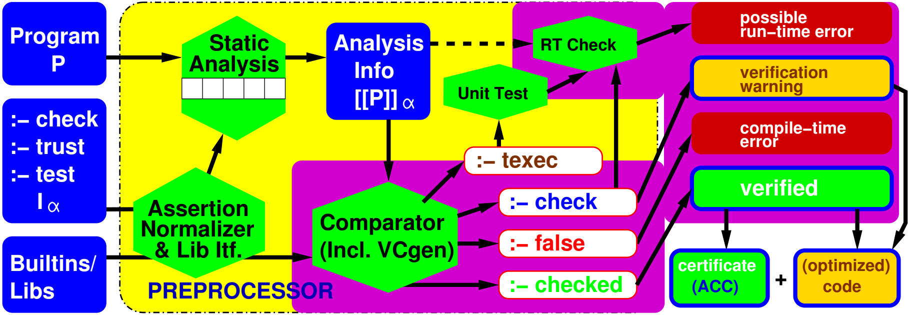


- Modular, parametric, polyvariant abstract interpretation.
- Accelerated, incremental fixpoint.
- Properties:
  - Shapes, data sizes, sharing/aliasing, CHA, determinacy, exceptions, termination, ...
  - Resources (time, memory, energy, ...), (user-defined) resources.

[MLNH07] [MH92, BGH99, PH96, HPMS00, NMLH07] [MGH94, BCHP96, PH00, BdIBH+01, PCPH06, PCPH08]

[MH89, MH91, DLGH97, VB02, BLGH04, LGBH05, NBH06, MSHK07] [MLH08, MKSH08, MMLH+08, MHKS08, MKH09, LG

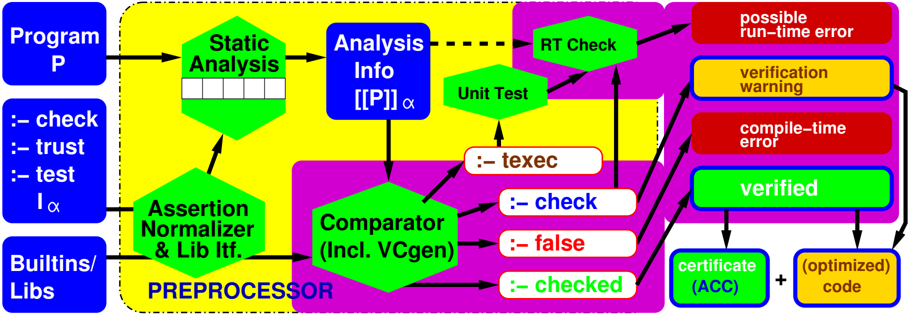
Integrated Static/Dynamic Debugging and Verification



	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]

Integrated Static/Dynamic Debugging and Verification

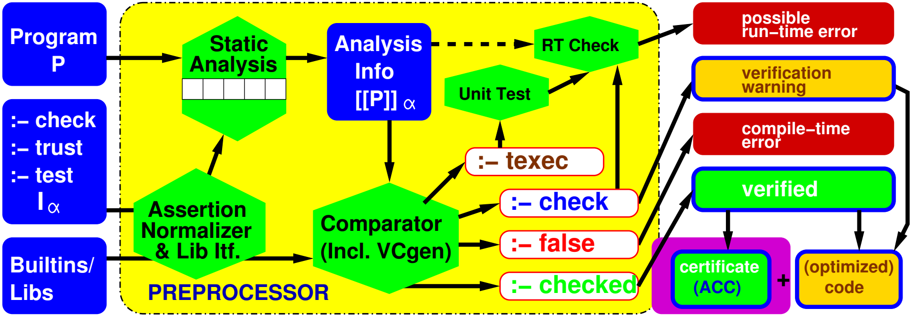


- Based throughout on the notion of *safe approximation* (abstraction).
- Run-time checks generated for *parts* of asserts. not verified statically.
- Diagnosis (for both static and dynamic errors).
- Comparison not always trivial: e.g., resource debugging/certification
  - Need to compare functions.
  - "Segmented" answers.

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

Demo: assertions, static errors (types, data sizes, procedure cost, non-determinacy, ...), run-time check generation, certification, unit tests...

Abstraction-based Certification, Abstraction-Carrying Code



PRODUCER | CONSUMER

$[\![P]\!]_\alpha$ = Analysis = **lfp**(*analysis_step*)

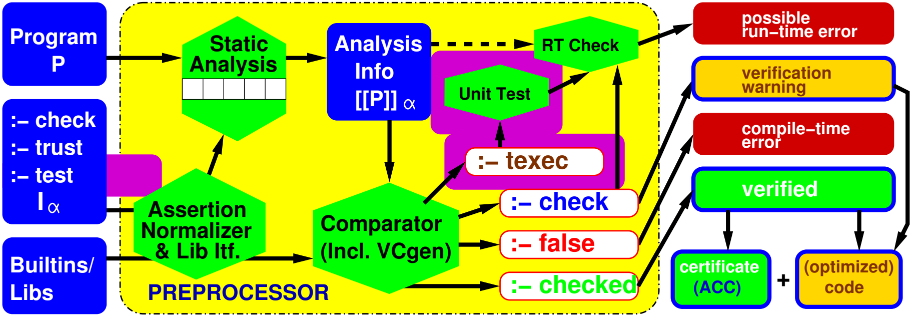
Certificate $\subset [\![P]\!]_\alpha$
Certificate →
Safety Policy

Checker = *analysis_step*

- Interesting extensions: reduced certificates, incrementality, ...

[APH05, HALGP05, AAPH06]

Integration of Testing

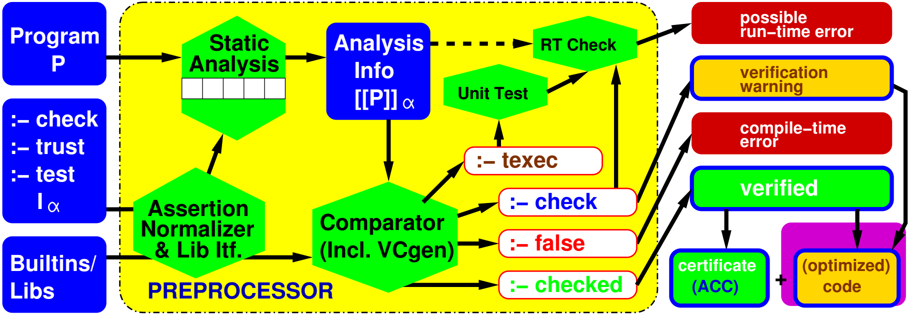


Many interactions within the integrated framework:

- (Unit) tests are part of the assertion language:
  `:- test Pred [:Precond] [=>Postcond] [+CompExecProps].`
- Parts of unit tests that can be verified at compile-time are deleted.
- Unit testing uses the run-time assertion-checking machinery.
- Unit tests also provide test cases for the run-time checks.
  - Assertions checked by unit testing, even if not conceived as tests.

[MLGH09]

Optimization



- Source-level optimizations:
  - Partial evaluation, (multiple) (abstract) specialization, ...
- Low-level optimizations (e.g., dynamic check elimination, unboxing):
  - Use of specialized instructions.
  - Optimized native code generation.

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

- Parallelization. Granularity control.

[GH91, PH97, PH03, PHG99, PAH06] [PH99, MBdlBH99, BGH99, CCH08, MKSH08] [MCH04, CMM+06]

Discussion: The Ciao Approach [AADEBUG'97, etc.]

- Approaches prior to Ciao had what we perceived as limitations:
  - limited the properties which may appear in specifications, or
  - checked specifications only at run-time or only at compile-time, or
  - were not automatic, or required assertions for all predicates, or ...
- The Ciao approach – solution to static/dynamic conundrum, which:
  - Integrates automatic compile-time and run-time checking of assertions.
  - Allows using assertions in only some parts of the program.
  - Deals *safely* with complex properties (beyond, e.g., traditional types).

  Allows "modern" (agile/extreme/...) programming, "Scripts to Ps:"
  - 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.
- Assertion language design is important: many roles, used throughout.
- Assertions, properties in source language; "seamless integration."
- Performance through optimization, not language restriction.

Discussion: Comparison with *Classical* Types

"Traditional" Types	**Ciao Assertion-based Model**
"Properties" limited by decidability	Much more general property language
May need to limit prog. lang.	No need to limit prog. lang.
"Untypable" programs rejected	Run-time checks introduced
(Almost) Decidable	Decidable + Undecidable(approximated)
Expressed in a different language	Expressed in the source language
Types must be defined	Types can be defined or inferred
Assertions are only of type "check"	"check", "trust", ...
Type signatures & assertions different	Type signatures *are* assertions

- Some key issues:
  Safe / Sound approximation — *Suitable assertion language*
  Abstract Interpretation — *Powerful abstract domains*
- Works best when properties and assertions can be expressed in the source language (i.e., source lang. supports *predicates*, *constraints*).

Outline

Part I The Ciao approach to Analysis and verification *of Constraint Logic Programs*

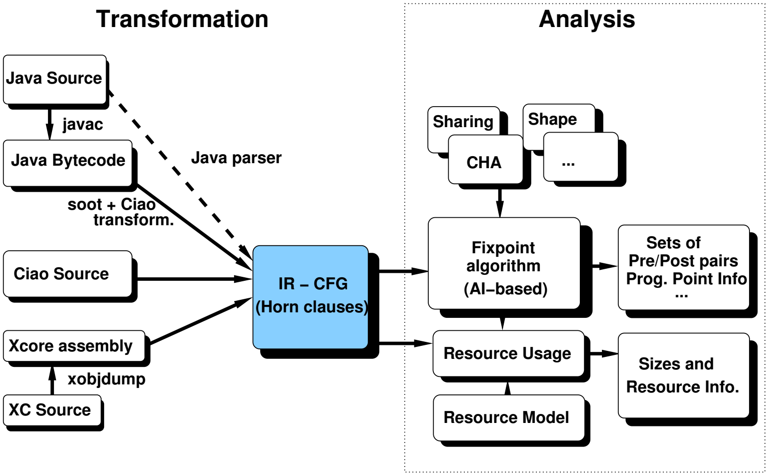
- The programming language
- The analysis, verification, and testing model

Part II The Ciao approach to Analysis and verification *of other paradigms using Constraint Logic Programs as IR*

- CLP (Horn Clauses) as intermediate representation
- User-defined resource analysis/verif. of Java bytecode
- Energy analysis/verification of (Xmos) C programs

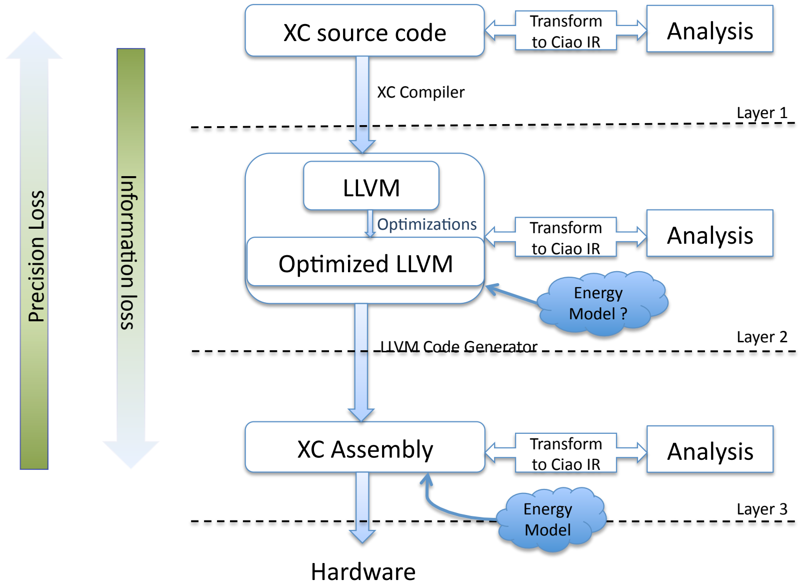
Intermediate Representation: (Constraint) Horn Clauses

[MLNH07]



- Allows supporting multiple languages / paradigms.
- Used for all analyses: aliasing, CHA/shape/types, data sizes / resources, etc.
- Based on "blocks:" each block represented as a *Horn clause*.

IR Issues: IR Level Trade-offs



IR Issues: Approaches to Performing the Transformation

- The transformation (akin to *Abstract Compilation*):
  - Source: Program P in L_P + (possibly abstract) Semantics of L_P
  - Target: A (C) Horn Clause program capturing the semantics of P

- Some approaches to performing the transformation:
  - Direct transformation into block-based intermediate representation.
    - More control but correctness proof more indirect.
    - Used in the following (translation to a Ciao program).
    - Can add assertions to help analysis (sizes, metrics, resource models, ..).
  - Partial evaluation of instrumented interpreters + slicing.
    - Systematic construction from small- and big-step semantics.
    - Correctness proof more direct.
    - Less automatic?

  Some evidence that the two approaches can produce similar results.

- Cf. John Gallagher's talk!

Generating the Intermediate Representation

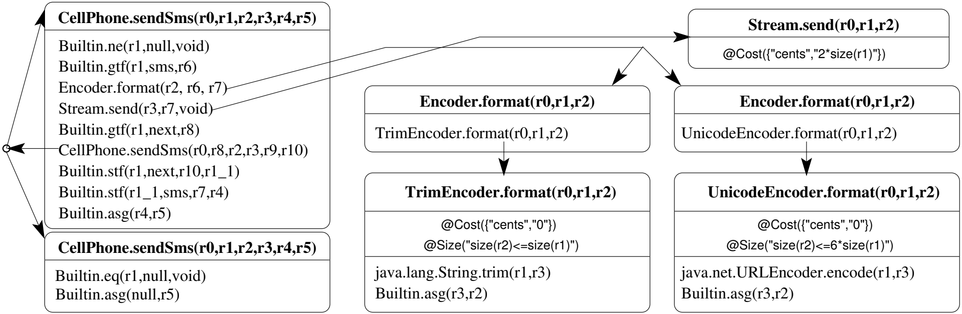
- Specifics for Java:
  - Control flow graph construction from bytecode representation.
  - Elimination of stack variables.
  - Conversion to three-address statements.
  - Explicit representation of this and ret as extra block parameters.
- Specifics for XC:
  - Control flow graph construction from ISA (or LLVM IR) representation.
  - Resolving branching to predicates with multiple clauses.
  - Inferring block parameters.
- Some common tasks:
  - Generation of block-based CFG.
  - SSA transformation (e.g., splitting of input/output param).
  - Conversion of loops into recursions among blocks.
  - Branching, cases, dynamic dispatch → blocks w/same signature.
  - Conversion to horn clauses.

Java Example 1: 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)
}}
```

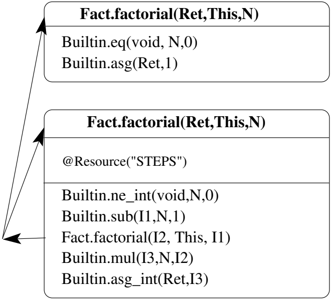
Java Example 1: sending SMSs – IR



- 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.

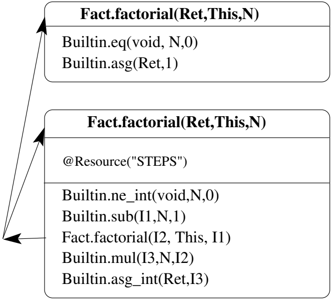
Java Example 2: Factorial

```
@Resources({Resource.STEPS})
public class Fact
{
  public int factorial(int n) {
    if (n == 0)
     return 1;
    else
     return n * factorial(n - 1);
  }
}
```



Source code → Basic blocks.

Java Example 2: Factorial



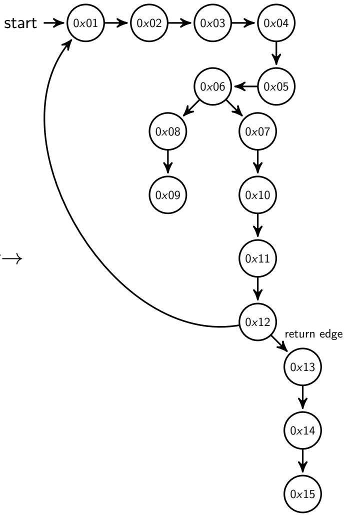

```
:- entry 'Fact.factorial'/3:var*atm*num.
:- resource 'STEPS'.
'Fact.factorial'(Ret, This, N):-
        eq_int(void,N,int,0,int),
        asg_int(Ret,int,1,int).

'Fact.factorial'(Ret, This, N):-
        ne_int(void,N,int,0,int),
        sub(I1,int,N,int,1,int),
        Fact.factorial(I2,This,I1),
        mul(I3, int,N,int,I2,int),
        asg_int(Ret,int,I3,int).
```

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

Xcore Example: Control Flow Graph (CFG)

```
<fact>:
0x01: entsp (u6)    0x2
0x02: stw (ru6)     r0, sp[0x1]
0x03: ldw (ru6)     r1, sp[0x1]
0x04: ldc (ru6)     r0, 0x0
0x05: lss (3r)      r0, r0, r1
0x06: bf (ru6)      r0, 0x1 <0x08>
0x07: bu (u6)       0x2 <0x10>
0x08: mkmsk (rus)   r0, 0x1
0x09: retsp (u6)    0x2
0x10: ldw (ru6)     r0, sp[0x1]
0x11: sub (2rus)    r0, r0, 0x1
0x12: bl (u10)      -0xc <fact>
0x13: ldw (ru6)     r1, sp[0x1]
0x14: mul (l3r)     r0, r1, r0
0x15: retsp (u6)    0x2
```

→Interpreter→



Block Representation

Basic block

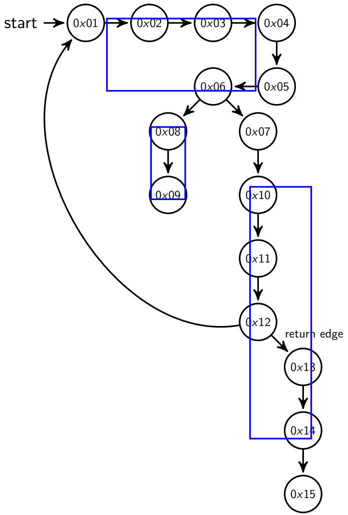
A basic block is a maximal sequence S of consecutive nodes G in CFG, starting from node n and ending in node m such that:

$$(\forall k \in S/\{n, m\}.\ outEdges(k) = 1 \wedge inEdges(k) = 1) \wedge$$
$$outEdges(n) = 1 \wedge inEdges(m) = 1$$

- Initial block starts from the entry node.
- Dead code elimination.

Xcore Example: Block Representation

```
<fact>
0x01: entsp (u6)    0x2
0x02: stw (ru6)     r0, sp[0x1]
0x03: ldw (ru6)     r1, sp[0x1]
0x04: ldc (ru6)     r0, 0x0
0x05: lss (3r)      r0, r0, r1
0x06: bf (ru6)      r0, 0x1 <0x08>

0x07: bu (u6)       0x2 <0x10>
0x10: ldw (ru6)     r0, sp[0x1]
0x11: sub (2rus)    r0, r0, 0x1
0x12: bl (u10)      -0xc <fact>
0x13: ldw (ru6)     r1, sp[0x1]
0x14: mul (l3r)     r0, r1, r0
0x15: retsp (u6)    0x2

0x08: mkmsk (rus)   r0, 0x1
0x09: retsp (u6)    0x2
```



Xcore Example: Block Representation

```
 fact :-
0x01: entsp(0x2)
0x02: stw(r0, sp[0x1])
0x03: ldw(r1, sp[0x1])
0x04: ldc(r0, 0x0)
0x05: lss(r0, r0, r1)
0x06: bf(r0, 0x1 <0x08>)
 branch(bf0, bf1)

 bf1 :-
0x07: bu(0x2 <0x10>)
0x10: ldw(r0, sp[0x1])
0x11: sub(r0, r0, 0x1)
0x12: bl(-0xc <fact>)
 call(fact)
0x13: ldw(r1, sp[0x1])
0x14: mul(r0, r1, r0)
0x15: retsp(0x2)

 bf0 :-
0x08: mkmsk(r0, 0x1)
0x09: retsp(0x2)
```

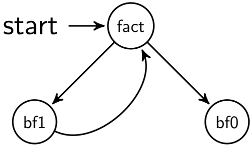
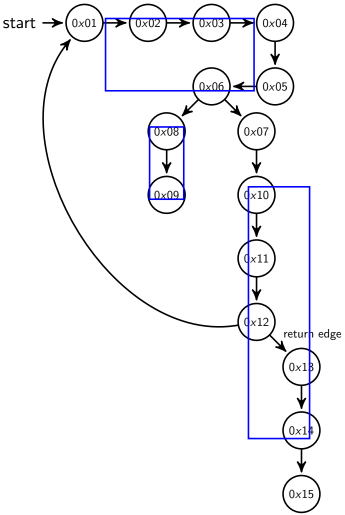


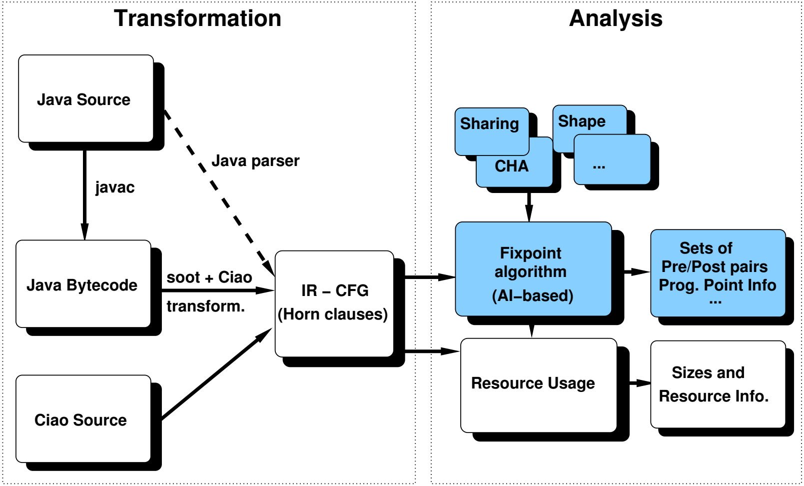
Figure: Block Control Flow Graph

Xcore Example: Horn Clause IR




```
:- entry fact/2 : int * var.
fact(R0,R0_3):-
  entsp(0x2),
  stw(R0,Sp0x1),
  ldw(R1,Sp0x1),
  ldc(R0_1,0x0),
  lss(R0_2,R0_1,R1),
  bf(R0_2,_),
  bf01(R0_2,Sp0x1,R0_3,R1_1).

bf01(1,Sp0x1,R0_4,R1):-
  bu(_),
  ldw(R0_1,Sp0x1),
  sub(R0_2,R0_1,0x1),
  bl(_),
  fact(R0_2,R0_3),
  ldw(R1,Sp0x1),
  mul(R0_4,R1,R0_3),
  retsp(0x2).

bf01(0,Sp0x1,R0,R1):-
  mkmsk(R0,0x1),
  retsp(0x2).
```

Fixpoint-based Analyzers



[MH92, BGH99, PH96, HPMS00, NMLH07] [MGH94, BCHP96, PH00, BdlBH+01, PCPH06, PCPH08]
[MH89, MH91, DLGH97, VB02, BLGH04, LGBH05, NBH06, MSHK07]
[MLH08, MKSH08, MMLH+08, MHKS08, MKH09, LGBH10, MLLH08] [SLBH13, LKSGL13, SLH13]

Efficient, Parametric Fixpoint Algorithm

- *Generic framework* for implementing analyses: given abstract domain, computes $\mathrm{lfp}(S_P^\alpha) = [\![P]\!]_\alpha$, s.t. $[\![P]\!]_\alpha$ safely approximates $[\![P]\!]$.
- It maintains and computes as a result (simplified):
  - **A call-answer table**: with (multiple) entries $\{block : \lambda_{in} \mapsto \lambda_{out}\}$.
    - 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, widening, ...
  - Handles mutually recursive methods.
  - Modular and *incremental*.
  - Handles library calls, externals, ...

Essentially efficient, incremental, (abstract) OLDT resolution.

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 / recursion support

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

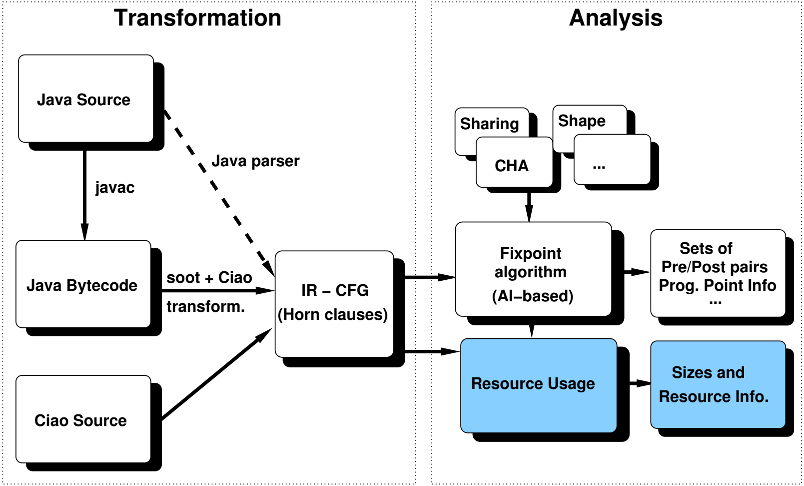
... abstractly execute each compatible block, ...

- 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).

Resource Analysis



[DLH90, LGHD94, LGHD96, DLGHL94, DLGHL97, NMLGH07, MLNH07, MLGCH08, NMLH08]

[NMLH09, LGDB10, SLBH13, LKSGL13, SLH13]

Analysis/Debugging/Verification of Resources

Automatically infer upper/lower bounds on the usage that a program makes of a general notion of various (*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., instructions, bytecodes, libraries).
  2. System infers the resource usage bounds for rest of program as **functions of input data sizes**.
- Involved properties normally undecidable → approximation required (**bounds** that are safe and also as accurate as possible).
- Applications: performance debugging and verification, resource-oriented optimization, granularity control in parallelism, ...

[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.

The Assertion Language (*simplified* grammar, Java)

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

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 ($\backslash uxxxx$) 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"	$\lambda n, m.n + k$	length	292
	"insertions Stores"	$\lambda n, m.n$		
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"	30232844295713061	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]

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

Resource Analysis as an Abstract Interpretation

[SLH13, SLBH13]

- In the classical CiaoPP resource analysis the last steps (setting up and solving recurrences) were not implemented as an abstract domain.
- We have now defined, implemented and integrated the resource analysis as an *abstract domain* (a plugin of the generic fixpoint).
- We get all the good features of the AI framework for free:
  - Multivariance: e.g., separate different call patterns for same block:
    `sort(lst(int),var) ...  sort(lst(flt),var) ...  sort(var,lst(int))`
  - Easier combination with other domains.
  - Easier integration w/static debugging/verification and rt-checking.
  - Many other engineering advantages.
- New domain for size analysis (*sized types*) that infers bounds on the size of data structures *and substructures*.
  - Size: number of rule applications in type/shape definition.
- Used in the XC energy analysis.

The Sized Types Abstract Domain

[SLBH13]

Sized types are representations of data shape information including both lower and upper bounds on the size of the corresponding terms and their subterms at any position and depth.

- Derived from the *regular types* inferred for program variables.
- If τ is a regular type, *sized*(τ) is its corresponding sized type:

listnum	*sized*(*listnum*)
`listnum -> []` `listnum -> [num | listnum]`	$listnum^{(\alpha,\beta)}\left(num^{(\gamma,\delta)}_{\langle .,1\rangle}\right)$

- The superscripts (*size bound variables*) express bounds on the number of rule (functor) applications.

  `{ [1,2,3,4], [2,4] }` $\quad listnum^{(3,5)}\left(num^{(1,4)}_{\langle .,1\rangle}\right)$

- Size analysis infers *relations (inequations)* among the *size bound variables* of the sized types occurring at different argument positions.

Experimental Results

Prog.	Resource An. (LB) New	Resource An. (LB) Prev.		Resource An. (UB) New	Resource An. (UB) Prev.		Resource An. (UB) RAML		An. Time (s) New	An. Time (s) Prev.
`append`	α	α	$=$	β	β	$=$	β	$=$	1.00	0.53
`appAll`	$a_1 a_2 a_3$	a_1	$+$	$b_1 b_2 b_3$	∞	$+$	$b_1 b_2 b_3$	$=$	2.41	0.67
`coupled`	μ	0	$+$	ν	∞	$+$	ν	$=$	1.37	0.64
`dyade`	$\alpha_1 \alpha_2$	$\alpha_1 \alpha_2$	$=$	$\beta_1 \beta_2$	$\beta_1 \beta_2$	$=$	$\beta_1 \beta_2$	$=$	1.66	0.62
`erathos`	α	α	$=$	β^2	β^2	$=$	β^2	$=$	2.25	0.77
`fib`	ϕ^μ	ϕ^μ	$=$	ϕ^ν	ϕ^ν	$=$	infeas.	$+$	1.06	0.67
`hanoi`	1	0	$+$	2^ν	∞	$+$	infeas.	$+$	0.82	0.60
`isort`	α^2	α^2	$=$	β^2	β^2	$=$	β^2	$=$	1.68	0.62
`isortl`	a_1^2	a_1^2	$=$	$b_1^2 b_2$	∞	$+$	$b_1^2 b_2$	$=$	2.55	0.67
`lisfact`	$\alpha\gamma$	α	$+$	$\beta\delta$	∞	$+$	unkn.	?	1.39	0.64
`listnum`	μ	μ	$=$	ν	ν	$=$	unkn.	?	1.19	0.58
`minsort`	α^2	α	$+$	β^2	β^2	$=$	β^2	$=$	1.94	0.67
`nub`	a_1	a_1	$=$	$b_1^2 b_2$	∞	$+$	$b_1^2 b_2$	$=$	3.61	0.91
`part`	α	α	$=$	β	β	$=$	β	$=$	1.70	0.65
`zip3`	$\min(\alpha_i)$	0	$+$	$\min(\beta_i)$	∞	$+$	β_3	$+$	2.48	0.57

Energy Consumption Analysis

- Specialize the generic resource analysis by encoding energy models: provide cost and size assertions for each individual instruction.
- Some energy models:
  - Java bytecode energy consumption models available for simple processors –upper bound consumption per bytecode in 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
...	...	...	...

  - More sophisticated *ISA-level energy models* developed w/Bristol & XMOS (based on "Tiwari" model).
- The CiaoPP resource analysis then generates at compile time safe upper- and lower-bound *energy consumption functions* for given programs.

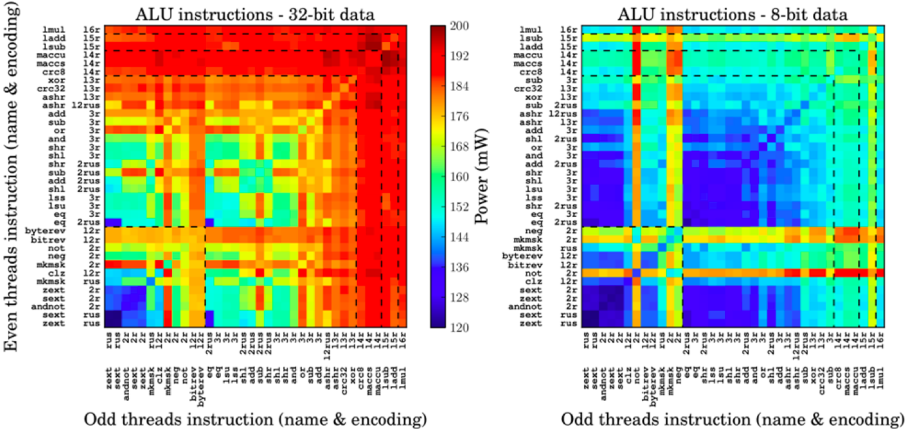
[NMLH08]

Demo: java resource analysis (including CHA, nullity, etc.);

XC energy analysis.

Low-level ISA characterization

Obtaining the cost model: energy consumption per instruction



Coll. w/Xmos and Bristol U (based on Tiwari model).

Energy Model

Expressed in the Ciao assertion language

```
:- package(energy).
:- use_package(library(resources(definition))).
:- load_resource_definition(ciaopp(xcore(model(res_energy)))).

:- trust pred mkmsk_rus2(X)
        : var(X) => (num(X), rsize(X,num(A,B)))
        + ( resource(energy, 1112656, 1112656) ).

:- trust pred add_2rus2(X)
        : var(X) => (num(X), rsize(X,num(A,B)))
        + ( resource(energy, 1147788, 1147788) ).

:- trust pred add_3r2(X)
        : var(X) => (num(X), rsize(X,num(A,B)))
        + ( resource(energy, 1215439, 1215439 )).

:- trust pred sub_2rus2(X)
        : var(X) => (num(X), rsize(X, num(A,B)))
        + ( resource(energy, 1150574, 1150574)).

:- trust pred sub_3r2(X)
        : var(X) => (num(X), rsize(X,num(A,B)))
        + ( resource(energy, 1210759, 1210759 )).

:- trust pred ashr_l2rus2(X)
        : var(X) => (num(X), rsize(X,num(A,B)))
        + ( resource(energy, 1219682, 1219682) ).
--:---  energy.pl      Top L1     (Ciao)----------------------------------------
```

XC Source

fact.xc

```
#include "fact.h"

int fact(int i) {
  if(i<=0)  return 1;
  return i*fact(i-1);
}
```

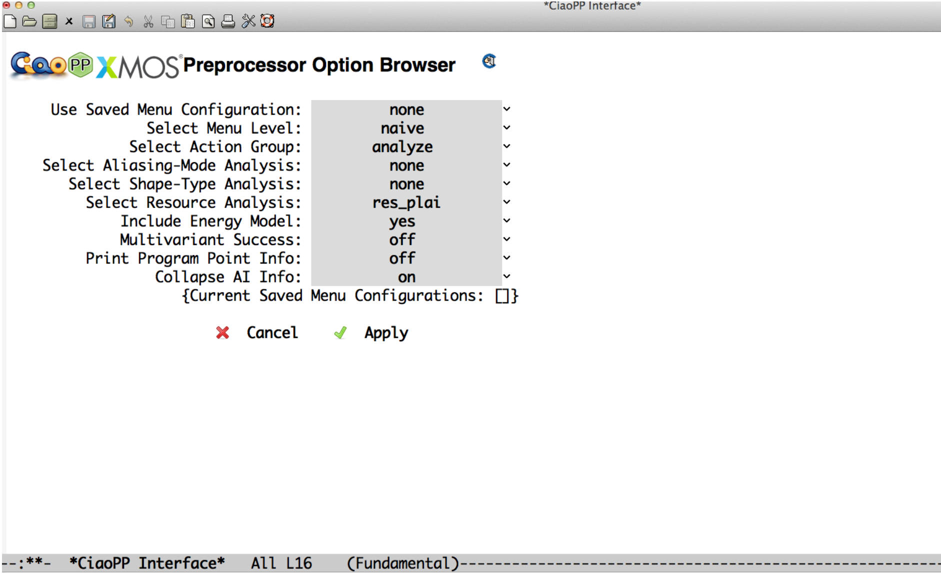
```
--:---  fact.xc        All L10    (C/l Abbrev)----------------------------------------
<nil> <drag-mouse-1> is undefined
```

Assembly Code

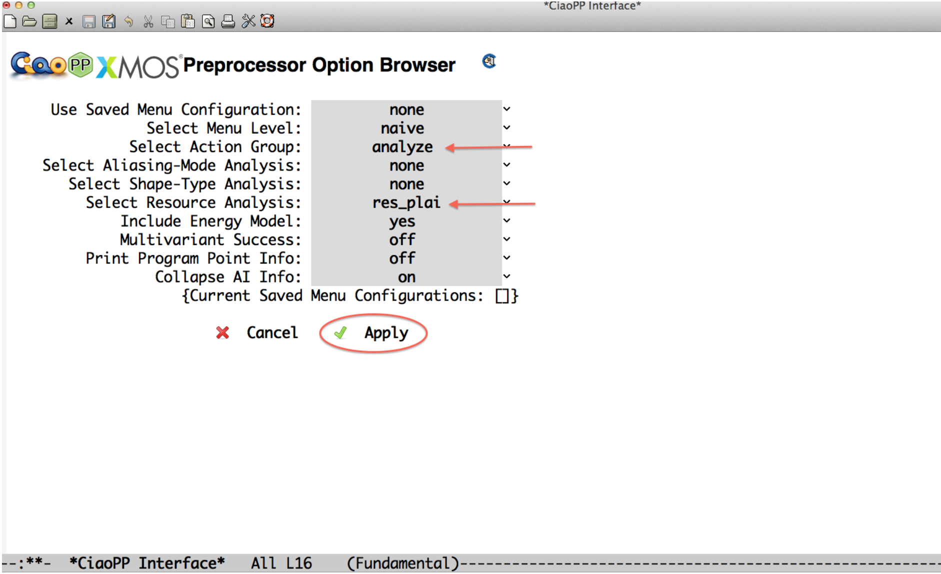
factassembly.pl

```
fact:
        entsp 6
        stw r0, sp[4]
        stw r0, sp[2]
.Lxtalabel0:
        ldw r0, sp[4]
        ldc r1, 0
        lss r0, r1, r0
        bt r0, .LBB0_4
        bu .LBB0_3
.LBB0_3:
        mkmsk r0, 1
        stw r0, sp[3]
        bu .LBB0_5
.LBB0_4:
.Lxtalabel1:
        ldw r0, sp[4]
        sub r1, r0, 1
        stw r0, sp[1]
        mov r0, r1
.Lxta.call_labels0:
        bl fact
        ldw r1, sp[1]
        mul r0, r1, r0
        stw r0, sp[3]
.LBB0_5:
        ldw r0, sp[3]
        retsp 6
--:---  factassembly.pl   Top L3    (Ciao)----------------------------------------
```

CiaoPP Menu


CiaoPP Interface
Preprocessor Option Browser
Use Saved Menu Configuration: none
Select Menu Level: naive
Select Action Group: analyze
Select Aliasing-Mode Analysis: none
Select Shape-Type Analysis: none
Select Resource Analysis: res_plai
Include Energy Model: yes
Multivariant Success: off
Print Program Point Info: off
Collapse AI Info: on
{Current Saved Menu Configurations: []}
Cancel
Apply
--:**- *CiaoPP Interface* All L16 (Fundamental)----------

Select Resource Analysis



Analysis Results

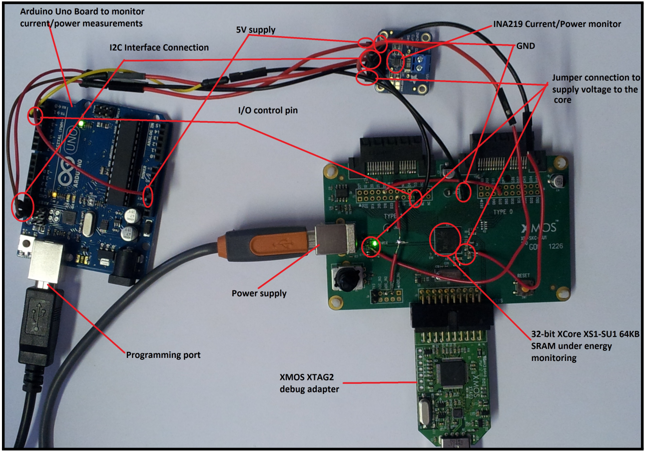
fact_results.pl

```
:- module(_,[fact/2],[ciaopp(xcore(model(instructions))),ciaopp(xcore(model(energy))),assertions]).

:- true pred fact(X,Y)
         : ( num(X), var(Y) )
        => ( num(X), num(Y), rsize(X,num(A,B)), rsize(Y,num('Factorial'(A),'Factorial'(B))) )
         + ( resource(energy, 6439360, 21469718 * B + 16420396) ).

fact(X,Y) :-
        entsp_u62(_3459),
        _3467 is X,
        stw_ru62(_3476),
        _3484 is X,
        stw_ru62(_3493),
        _3501 is _3467,
        ldw_ru62(_3510),
        _3518 is 0,
        ldc_ru62(_3527),
        _3518<_3501,
        lss_3r2(_3544),
        bt_ru62(_3552),
        1\=0,
        _3569 is _3467,
        ldw_ru62(_3578),
        _3586 is _3569-1,
        sub_2rus2(_3598),
        _3606 is _3569,
        stw_ru62(_3615),
        _3623 is _3586+0,
--:---  fact_results.pl    Top L11    (Ciao)----------------------------------------------
```

Checking against actual HW energy consumption

Test programs run on *two different* HW rigs:

- ISS (Instruction Set Simulation) and
- SRA (Static Resource Analysis).

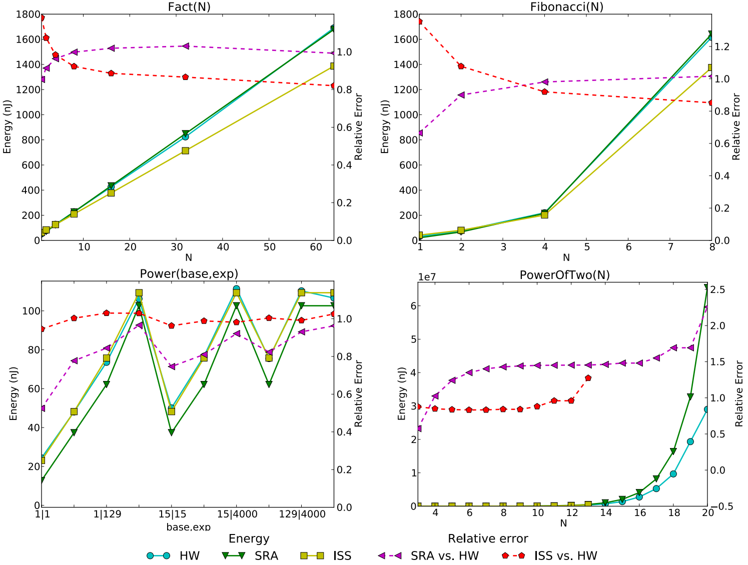


Some Results

Benchmarks

Function name	Description	Energy function
`fact(N)`	Calculates $N!$	$26.0\ N + 19.4$
`fibonacci(N)`	Nth Fibonacci no.	$30.1 + 35.6\ \phi^N + 11.0\ (1-\phi)^N$
`sqr(N)`	Computes N^2	$103.0\ N^2 + 205.8\ N + 188.32$
`poweroftwo(N)`	Calculates 2^N	$62.4 \cdot 2^N - 312.3$
`sumofdigits(N)`	Adds all digits in N	$84.4\lceil \log_{10} N \rceil - 78.7$
`isprime(N)`	Checks if N is prime	$58.6\ N - 35.5$
`power(base,exp)`	Calculates $base^{exp}$	$6.3\ (\log_2 exp + 1) + 6.5$

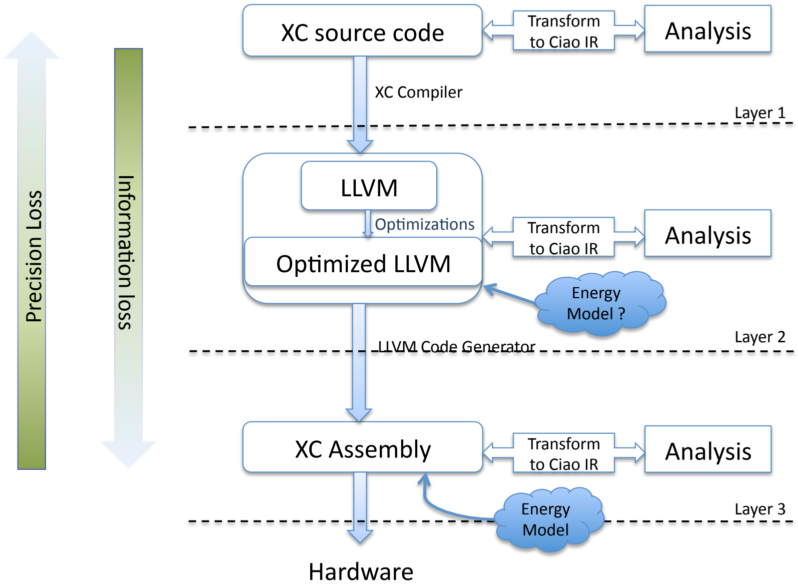
Some Results



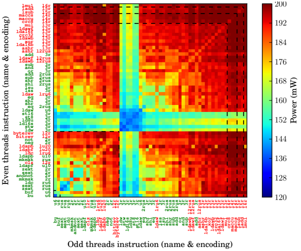
Feedback from the hardware experts (Xmos, Bristol)

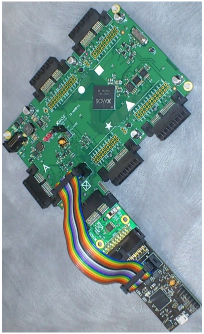
- SRA provides results *beyond what is possible with simulation* (as test run-time increases, ISS becomes impractically long).
- SRA shows promising accuracy in comparison with ISS and the HW (at least for the simple cases studied so far).
- Simulation time limits the usefulness of ISS method, whereas equation solving limits SRA.

IR Level Trade-offs



LLVM IR vs. ISA tradeoff





xC Program	Error vs. HW		ISA / LLVM IR
	llvm	isa	
fact	4.5%	2.86%	0.94
fibonacci	11.94%	5.41%	0.92
sqr	9.31%	1.49%	0.91
power_of_two	11.15%	4.26%	0.93
reverse	2.18%	N/A	N/A
concat	8.71%	N/A	N/A
mat_mult	1.47%	N/A	N/A
sum_facts	2.42%	N/A	N/A
Average	**6.46%**	**3.50%**	**0.92**

Energy consumption verification / debugging

```
:- check pred fact(A, B) :  (int(A), var(B))
                          + resource(energy, 0, 100).
```

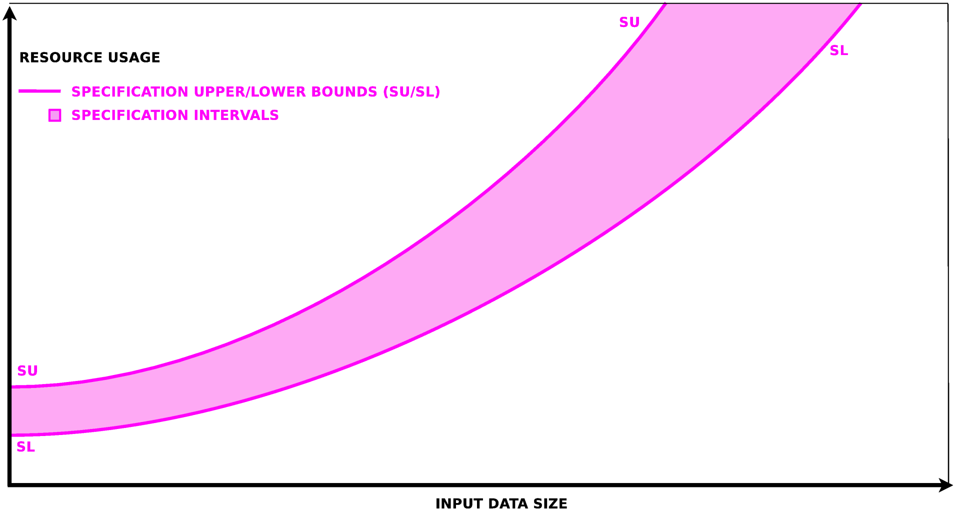
1. Resource analysis infers upper and lower bounds for resource "energy." The analysis results produced are:

```
:- true pred fact(A,B)
   :  (int(A), var(B))
   => (int(A), int(B), rsize(A, num(LA,UA)),
      rsize(B, num('Factorial'(LA),'Factorial'(UA))))
   + resource(energy, 21 * LA + 16, 21 * UA + 16).
```

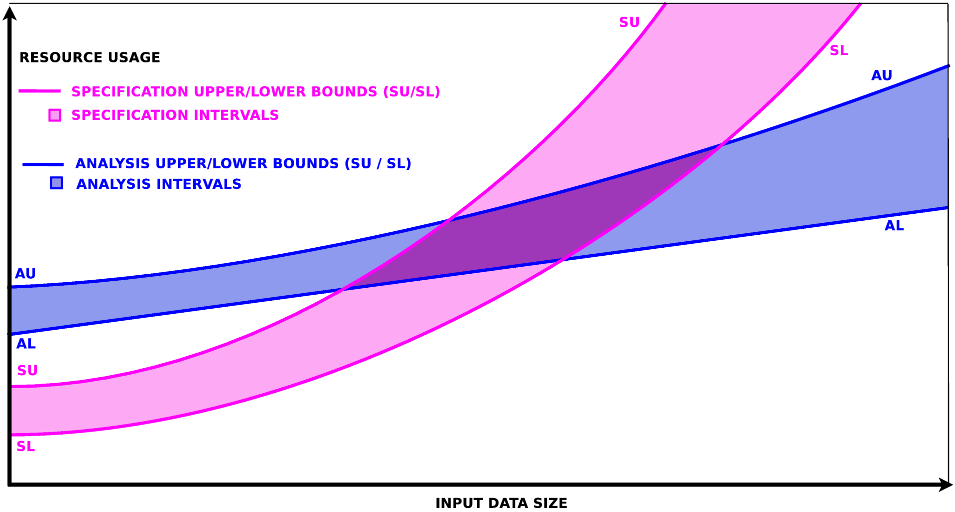
2. Then, the analysis results are compared with the "check" assertion (the specification) and the following assertions are produced:

```
:- checked pred fact(A, B)
   :  (int(A), intervals(int(A), [i(0,4)]), var(B))
   + resource(energy, 0, 100).
:- false pred fact(A, B)
   :  (int(A), intervals(int(A), [i(5,inf)]), var(B))
   + resource(energy, 0, 100).
```

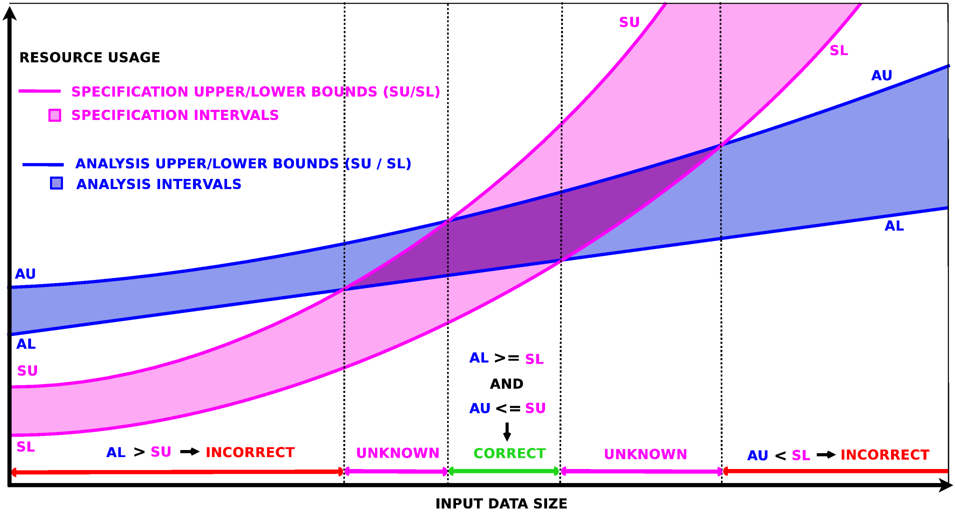
Resource Usage Verification – Function Comparisons



Resource Usage Verification – Function Comparisons



Resource Usage Verification – Function Comparisons



Tools / timeline

'83	Parallel abstract machines → motivation: auto-parallelization.
'88	**MA3 analyzer**: memo tables (cf. OLDT resolution), practicality established.
'89	**PLAI framework**: accelerated fixpoint, abstract domains as plugins. Sharing analysis, side-effect analysis.
90's	Incremental analysis, concurrency (dynamic scheduling), automatic domain combinations, scalability, auto-parallelization, extension to constraints.
'90	**GraCos analyzer**: fully automatic cost analysis (upper bounds).
early 90's	Automatic parallelization with task granularity control.
mid 90's	**Ciao model**: *Integrated verification/debugging/optimization w/assertions.*
'97-present	**CiaoPP tool**:
'91-'06	Combined abstract interpretation and partial evaluation.
late 90's	*Lower bound* cost analysis. Non-failure (no exceptions), determinacy.
'01	*Verification* of cost, additional resources, ...
'01-05	Modularity/scalability. Diagnosis (locating origin of assrt. violations). New shape/type domains, widenings. Polyhedra, convex hulls.
'03	Abstraction carrying code, reduced certificates.
'04	Verification/debugging/optimization of *user-defined* resources.
'05	*Multi-language support* using CLP as IR: Java, C# (shapes, resources, ...).
'08	Verification of exec. time. First results in energy (Java), heap models, ...
'12	(X)C program energy analysis/verification, ISA-level energy models.
'13	*Cost analysis as abstract interpretation.* Sized shapes inference. LLVM.

http://www.ciao-lang.org

Provides access to:

- Ciao, CiaoPP, LPdoc, etc.
- Documentation.
- Mailing lists.
- etc.

Please contact us for GIT access.

Around 1,000,000 lines of (mostly Ciao/Prolog) code.

Mostly **LGPL** (some packages have some variations).

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 – Horn Clauses as 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

[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.

[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.

References – Fixpoint-based Framework (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.

[SLBH13] A. Serrano, P. López-Garcia, F. Bueno, M. Hermenegildo.
Sized Type Analysis Logic Programs (Technical Communication).
In *Theory and Practice of Logic Programming, 29th Int'l. Conference on Logic Programming (ICLP'13) Special Issue, On-line Supplement,* pages 1–14, Cambridge U. Press, August 2013.

[LKSGL13] U. Liqat, S. Kerrison, A. Serrano, K. Georgiou, P. López-Garcia, N. Grech, M.V. Hermenegildo, K. Eder.
Energy Consumption Analysis of Programs based on XMOS ISA-Level Models.
In *Pre-proceedings of the 23rd International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR'13)*, September 2013.

[SLH13] A. Serrano, P. López-Garcia, M. Hermenegildo.
Towards an Abstract Domain for Resource Analysis of Logic Programs Using Sized Types.
In *23rd Workshop on Logic-based Methods in Programming Environments (WLPE 2013)*, 15 pages, August 2013.

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.

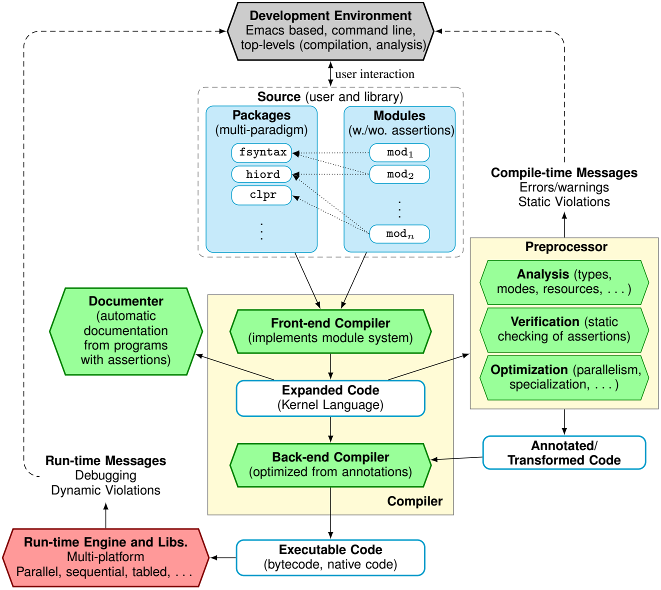
[CCH08] A. Casas, M. Carro, and M. Hermenegildo.
A High-Level Implementation of Non-Deterministic, Unrestricted, Independent And-Parallelism.
In M. García de la Banda and E. Pontelli, editors, *24th International Conference on Logic Programming (ICLP'08)*, volume 5366 of *LNCS*, pages 651–666. Springer-Verlag,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

Ciao Architecture Overview

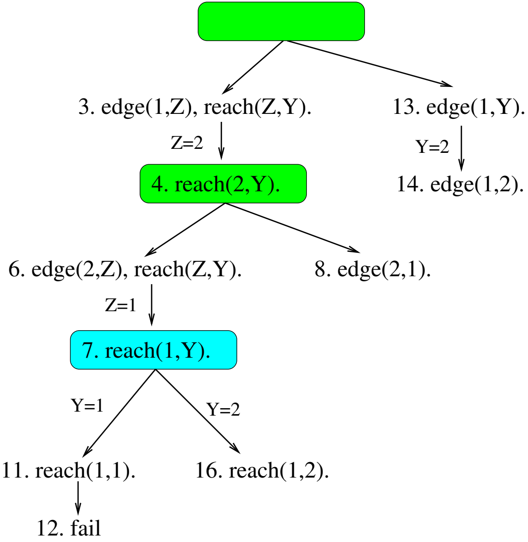


The tabling algorithm via an example (OLDT resolution)

Example

```
:- table reach/2.

reach(X,Y) :-
    edge(X,Z),
    reach(Z,Y).
reach(X,Y) :-
    edge(X,Y).

edge(1,2).
edge(2,1).

?- reach(1,Y).

Y = 1;
Y = 2;
no
```

Subgoal	Answers
2. reach(1,Y)	10. Y = 1 15. Y = 2 18. Complete
5. reach(2,Y)	9. Y = 1 17. Y = 2 18. Complete



Entailment vs. Call Abstraction

- TCHR: implementation of CHR on top of XSB Prolog with tabling.
  - It uses call abstraction.
- Reach: is some graph in a node reachable within some distance?

	Ciao TCLP	**TCHR**
Reach 30	7 140	129 978
Reach 25	6 680	129 876
Reach 20	5 964	128 955
Reach 15	4 316	129 313
Reach 10	2 296	128 994
Reach 5	427	129 616
Reach 0	1	129 472

- Constraints reduce the search space.

The Assertion Language (subset)

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

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

:- pred quicksort(X,Y) : list(int) * var => sorted(Y) + (is_det,not_fails).

:- pred quicksort(X,Y) : var * list(int) => ground(X) + non_det.

Properties; from libraries or user defined (in the source language):

:- regtype color := green | blue | red.
:- regtype list(X) :=[] | [X|list]. ≡ **list(_,[]). list(X,[H|T]) :- X(H), list(X,T).**

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

Types/shapes, cost, data sizes, aliasing, termination, determinacy, non-failure, ...

Program-point Assertions:

- Inlined with code: `..., check( int(X), X>0, mshare([[X]]) ), ....`

Assertion Status (so far "to be checked" – **check** status – default)

- Also: **trust** (guide analyzer), **true**/**false** (analysis output), **test**, etc.

Verification and Error Detection using 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]