

Computational Logic:
(Constraint) Logic Programming
Theory, practice, and implementation

Abstract Specialization
and its Applications

The following people have contributed to this course material:

*Manuel Hermenegildo (editor), Francisco Bueno, Manuel Carro, Germán Puebla, Pedro López, and Daniel Cabeza, Technical
University of Madrid, Spain*

Program Optimization and Specialization

- The aim of *program optimization* is, given a program P to obtain a program P' such that:
 - ◇ $\llbracket P \rrbracket = \llbracket P' \rrbracket$
 - ◇ P' behaves better than P for some criteria of interest.
- The aim of *program specialization* is, given:
 - ◇ a program P
 - ◇ and certain knowledge ϕ about the context in which P will be executed,to obtain a program P_ϕ such that
 - ◇ $\phi \Rightarrow \llbracket P \rrbracket = \llbracket P_\phi \rrbracket$
 - ◇ P_ϕ behaves better than P from some given point of view.

Some Techniques for Program Specialization

- Many techniques exist for program specialization which provide useful optimizations:
 - ◇ Partial Evaluation has received considerable attention and has been proved of interest in many contexts.
 - ◇ Supercompilation is another very powerful technique for program specialization.
 - ◇ Abstract Specialization (the topic of this talk!)

Partial Evaluation

- Its main features are
 - ◇ the knowledge ϕ which is exploited is the so-called *static* data, which corresponds to (partial) knowledge at specialization (compile) time about the input data.
 - ◇ Data which is not known at specialization time is called *dynamic*.
 - ◇ The program is optimized by performing at specialization time those parts of the program execution which only depend on static data.
- Has been applied to many programming languages and paradigms
- Two basic approaches have been addressed:
 - ◇ On-line techniques
 - ◇ Off-line techniques

Abstract Specialization

- We have proposed *Abstract Specialization*, whose main ingredients are:
 - ◇ Abstract Interpretation for
 - * allowing abstract information
 - * performing fixpoint computations for computing (safe approximations of) success substitutions.
 - ◇ Abstract Executability for optimizing program literals
 - ◇ Using the program induced by polyvariant abstract interpretation
 - ◇ Minimizing the number of versions
- It is a generic technique. Different abstract domains can be used which allow different information and optimizations.
- All four ingredients above are essential. Our system provides the first practical implementation of a system with all the features above.

Abstract Interpretation / Safe Approximation [CC77]

- *Abstract interpretation* obtains information about the run-time behavior of the program by
 - ◇ simulating the execution using an *abstract domain* which is simpler than the concrete one,
 - ◇ performing fixpoint computations for recursive calls.
- An *abstract domain* D_α is the set of all possible abstract semantic values.
- Values in the abstract domain are finite representations of a, possibly infinite, set of values in the concrete domain (D).

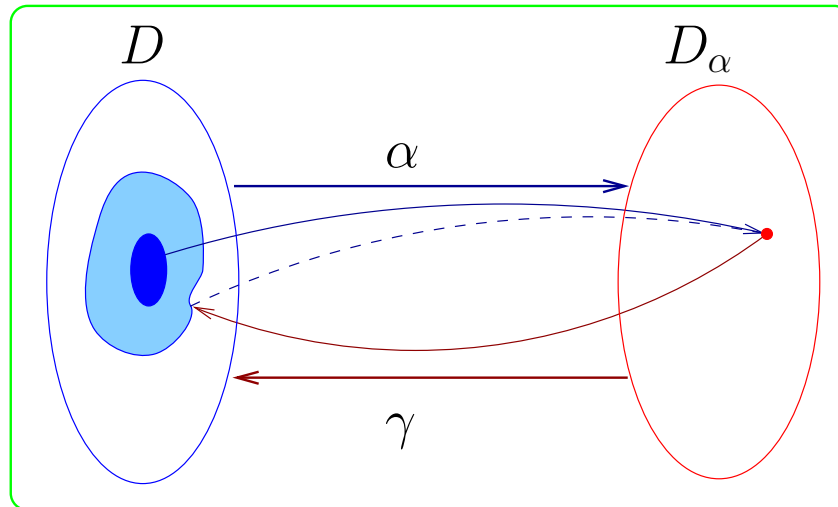
Abstract Interpretation (cont.)

- D and D_α are related via a pair of monotonic mappings $\langle \alpha, \gamma \rangle$:

- ◇ *abstraction* $\alpha : D \mapsto D_\alpha$

- ◇ *concretization* $\gamma : D_\alpha \mapsto D$

such that: $\forall x \in D : \gamma(\alpha(x)) \supseteq x$, and $\forall y \in D_\alpha : \alpha(\gamma(y)) = y$. i.e., $\langle \alpha, \gamma \rangle$ conform a *Galois insertion* over D and D_α .



A Simple Motivating Example

- Consider the Prolog program below

```
plus1(X,Y):- ground(X), Y is X + 1.
```

```
plus1(X,Y):- var(X), X is Y - 1.
```

- It defines the relation that the second argument is the first argument plus one.
- It can be used when at least one of its arguments is instantiated.
- Example executions:
 - ◇ The call `plus1(5, Result)` computes the addition of 1 and 5 and assigns it to `Result`.
 - ◇ The call `plus1(Num, 3)` determines which is the number `Num` such that when added to 1 returns 3 (i.e., $\text{Num} = 3 - 1$).
 - ◇ The call `plus1(3, 8)` fails.

Partial Evaluation of Our Simple Example

- Suppose we want to specialize the program below for the initial query `?- p(2,Res)`.

```
p(Value,Res):- Tmp is Value * 3, plus1(Tmp,Res).
```

```
plus1(X,Y):- ground(X), Y is X + 1.
```

```
plus1(X,Y):- var(X), X is Y - 1.
```

- Using an appropriate unfolding rule, a partial evaluator would compute the following specialized program:

```
p(2,7).
```

in which all computation has been performed at analysis time.

The need for Abstract Values

- Imagine we want to specialize the same program (repeated below) but now for the initial query `?- p(Value,Res)`.

```
p(Value,Res):- Tmp is Value * 3, plus1(Tmp,Res).
```

```
plus1(X,Y):- ground(X), Y is X + 1.
```

```
plus1(X,Y):- var(X), X is Y - 1.
```

- Since `Value` is dynamic, we cannot compute `Tmp` at specialization time.
- As a result, the call `plus1(Tmp,Res)` cannot be optimized.
- Using a sensible unfolding rule, the specialized program corresponds to the original one.

The Substitutions Computed by Abstract Interpretation

```
p(Value,Res) :-
    true(( term(Value), term(Res), term(Tmp) )),
    Tmp is Value * 3,
    true(( arithexpression(Value), term(Res), num(Tmp) )),
    plus1(Tmp,Res),
    true((arithexpression(Value), arithexpression(Res),num(Tmp)))

plus1(X,Y) :-
    true(( num(X), term(Y) )),
    ground(X),
    true(( num(X), term(Y) )),
    Y is X+1,
    true(( num(X), num(Y) )).

plus1(X,Y) :-
    true(( num(X), term(Y) )),
    var(X),
    true(( num(X), term(Y) )),
    X is Y-1,
    true(( num(X), arithexpression(Y) )).
```

Program Specialized w.r.t. Abstract Values

- The information above has been inferred with the regular types abstract domain *eterms*.
- Since $\text{num}(X) \Rightarrow \text{ground}(X)$, clearly the information available allows optimizing the program to:

```
p(Value,Res):- Tmp is Value + 3, Res is Tmp + 1.
```

- This optimization is not possible using traditional partial evaluation techniques.
- The use of abstract values allows capturing information which is simply not observable in traditional (on-line) partial evaluation.

Abstract Executability

- The optimization above is based on *abstract execution*
- A literal L in a program P can be reduced to the value
 - ◇ *true* if its execution can be guaranteed to succeed only once with the empty computed answer
 - ◇ *false* if its execution can be guaranteed to finitely fail.

Property	Definition	Sufficient condition
L is abstractly executable to <i>true</i> in P	$RT(L, P) \subseteq TS(L, P)$	$\exists \lambda' \in A_{TS}(\overline{B}, D_\alpha) : \lambda_L \sqsubseteq \lambda'$
L is abstractly executable to <i>false</i> in P	$RT(L, P) \subseteq FF(L, P)$	$\exists \lambda' \in A_{FF}(\overline{B}, D_\alpha) : \lambda_L \sqsubseteq \lambda'$

The Need for Approximating Success Substitutions

- Consider program below with query $?- p(\text{Res})$. and `plus1/2` is defined as usual. Note that the execution tree of `even/1` is infinite.

```
p(Res):- even(Tmp), plus1(Tmp,Res).
```

```
even(0).
```

```
even(E):- even(E1), E is E1 + 2.
```

- In order to compute some approximation of the success substitution for `even(Tmp)` a fixpoint computation is required.
- The information can be used to optimize the program as follows:

```
p(Res):- even(Tmp), Res is Tmp + 1.
```

```
even(0).
```

```
even(E):- even(E1), E is E1 + 2.
```

The Need for Polyvariant Specialization

- The approach just shown is simple and can achieve relevant optimizations
- However, opportunities for optimization quickly disappear if polyvariance is not used
- Consider the following program:

```
p(Value,Res):-  
    plus1(Tmp, Value), Tmp2 is Tmp * 3, plus1(Tmp2,Res).
```

- note that two different calls for plus1/2 appear.

The Need for Polyvariant Specialization (Cont.)

- The analysis information we now get is:

```
plus1(X,Y) :-  
    true(( term(X), term(Y) )),  
    ground(X),  
    true(( term(X), term(Y) )),  
    Y is X+1,  
    true(( arithexpression(X), num(Y) )).
```

```
plus1(X,Y) :-  
    true(( term(X), term(Y) )),  
    var(X),  
    true(( term(X), term(Y) )),  
    X is Y-1,  
    true(( num(X), arithexpression(Y) )).
```

- which no longer allows abstractly executing either of the two tests.

The Need for Polyvariant Specialization (Cont.)

- This can be solved by having two separate versions for predicate plus/3.

```
p(Value,Res):-  
    plus1_1(Tmp, Value), Tmp2 is Tmp + 3, plus1_2(Tmp2,Res)
```

```
plus1_1(X,Y):- ground(X), Y is X + 1.  
plus1_1(X,Y):- var(X),    X is Y - 1.
```

```
plus1_2(X,Y):- ground(X), Y is X + 1.  
plus1_2(X,Y):- var(X),    X is Y - 1.
```

The Need for Polyvariant Specialization (Cont.)

- which allows optimizing each version separately as

```
plus1_1(X,Y) :- X is Y-1.
```

```
plus1_2(X,Y) :- Y is X+1.
```

- And even unfolding the calls to `plus1/2` in the first clause, resulting in:

```
p(Value,Res) :-  
    Tmp is Value -1, Tmp2 is Tmp+3, Res is Tmp2 + 1.
```

Which Additional Versions Should We Generate?

- Though in the example above it seems relatively easy,
- It is not straightforward in general to decide when to generate additional versions.
- This problem is often referred to as “Controlling Polyvariance”.
- It has received considerable attention in different contexts: program analysis, partial evaluation, etc.
- We have to enable as many optimizations as possible, but we have to avoid:
 - ◇ code-explosion
 - ◇ or even non-termination!

Difficulties in Controlling Polyvariance: Spurious Versions

- Naive heuristics: generate one version per call to the predicate.
- Consider the following program:

```
p(Value,Res):-  
    plus1(Tmp, Value), plus1(Tmp,Tmp1), plus1(Tmp1,Tmp2),  
    plus1(Tmp2,Tmp3), plus1(Tmp3,Res).
```

- By generating multiple versions of `plus1/2` we can optimize them. Otherwise it is not possible.
- However, the *expanded* program would have 5 different versions for `plus1/2`, when in fact only two of them are needed.
- This shows that we have to avoid generating spurious versions.

Difficulties in Controlling Polyvariance: Insufficient Versions

- Consider the following program:

```
p(Value,Res):- q(Tmp, Value), Tmp2 is Tmp + 3, q(Tmp2,Res).
```

```
q(A,B):-
```

```
    % may do other things as well
```

```
    other(A,B), plus1(A,B).
```

```
other(A,B):- write(A), write(B).
```

- By using the naive approach only one version of `plus1/2` will be generated. As a result, it cannot be optimized.
- The solution lies in generating two versions of `q/2` in order to create a “path” to the now separately specialized versions.
- This shows that we may need more versions than calls in the program.

Difficulties in Controlling Polyvariance: Insufficient Versions

- The expanded program is shown below:

```
p(Value,Res):-  
    q_1(Tmp, Value), Tmp2 is Tmp + 3, q_2(Tmp2,Res).
```

```
q_1(A,B):- other(A,B), plus1_1(A,B).
```

```
q_2(A,B):- other(A,B), plus1_2(A,B).
```

- This program can now be optimized to:

```
p(Value,Res) :-  
    q_1(Tmp,Value), Tmp2 is Tmp+3, q_2(Tmp2,Res).
```

```
q_1(A,B) :- other(A,B), A is B - 1.
```

```
q_2(A,B) :- other(A,B), B is A + 1.
```

Polyvariant Abstract Interpretation

- *Goal-Dependent* abstract interpretation aims at computing both call and success abstract patterns
- In order to achieve as much accuracy as possible, most (goal-dependent) abstract interpreters are polyvariant.
- Different call patterns for the same procedure are treated separately.
- Different levels of polyvariance can be used with different degrees of accuracy and efficiency (trade-off)
- In spite of analysis being polyvariant, the usual approach is to “flatten” the information corresponding to different versions prior to using it to optimize the program.

Abstract Interpretation based on And–Or Graphs

- Many of the existing abstract interpreters for logic programs are based on an and–or tree semantics a la Bruynooghe.
- There, analysis builds an and–or graph in which:
 - ◇ *or–nodes* represent atoms (procedure calls). They are adorned with a call and a success substitution.
 - ◇ *and–nodes* represent clauses and are adorned with the head of the clause they correspond to.
- The and–or graph is often not explicitly computed for efficiency reasons.
- Thus, most optimizing compilers do not use the full and–or graph but rather a description of procedures as call and success pairs.
- This eliminates multiple specialization, losing opportunities for optimization.

The Polyvariant Program Induced by Abstract Interpretation

- Since polyvariant analysis considers several versions of procedures, this in effect corresponds to a polyvariant program:
 - ◇ Each pair (procedure, call pattern) corresponds to a different version.
 - ◇ The call pattern for each literal uniquely determines the version to use.
- Thus, the “polyvariant” program can be materialized.
- Each version is implemented with a different procedure name
- Renaming is performed so that no run-time overhead is incurred to select versions.

The Need for a Minimizing Algorithm

- Even for relatively small programs, the number of versions generated by analysis can be very large.
- It does not depend on whether the additional versions lead to additional optimizations or not.
- Our systems includes a minimizing algorithm which guarantees that:
 - ◇ the resulting program is minimal
 - ◇ while not losing any opportunities for specialization
- In our experience, using the minimization algorithm the size of the polyvariant program increases w.r.t. the original one, but does not explode.
- Other possibilities need to be explored.

DEMO: AUTOMATIC PARALLELIZATION

OPTIMIZATION OF DYNAMIC SCHEDULING

INTEGRATION WITH PARTIAL EVALUATION

Drawbacks of Traditional On-line PE (1/2)

- In on-line PE propagation of values and program transformation are performed simultaneously.
- Done by means of *unfolding*, which has two effects:
 - ◇ transforms the program,
 - ◇ propagates the values of variables computed during the transformation to other atoms in the node.
- Propagation can only be achieved through unfolding but:
 - ◇ Unfolding is not always a good idea.
 - ◇ Unfolding is not always possible.
The SLD tree may be infinite and we have to stop at some point.

Drawbacks of Traditional On-line PE (2/2)

- As soon as we stop unfolding, separate SLD trees are required.
- Thus, for most programs, several SLD trees exist.
- However, in PE there is no propagation between separate SLD trees.
- In summary, we would like to improve the propagation capabilities of PE by allowing:
 - ◇ propagation without transformation,
 - ◇ propagation among different SLD trees (residual atoms).

And–Or Graphs Vs. SLD Trees

- The advantages of And–Or graphs w.r.t. SLD trees are:
 - ◇ Propagation does not require transformation. Success substitutions are computed all over the graph.
 - ◇ Even infinite computations are approximated using fixpoint computations.
 - ◇ There is only one global graph and propagation among different atoms is possible and natural.
 - ◇ In addition to concrete values, it can also work with other interesting properties: types, modes, aliasing, etc.
- The advantages of SLD trees w.r.t. And–Or graphs are:
 - ◇ Simpler to build. No abstract domain, no fixpoint required.
 - ◇ Code-generation is straightforward.

Integrating PE into Abstract Interpretation

- In order to integrate PE into AI we need
 1. an abstract domain (and widening operator) which captures concrete values,
 2. an algorithm for code generation from the and-or graph,
 3. a way of performing unfolding steps when profitable.
- The first two points are fully achieved.
- Several alternatives exist for the third one (see the paper).
- The currently implemented approach is very simple.
- More powerful unfolding mechanisms have to be tested.

Related Work on Integration of PE and AI

- From a partial evaluation perspective, both in logic [GallagherCS88,Gallagher92] and functional programming [ConselK93].
- From an abstract interpretation perspective in [GiannottiH91] and the first complete framework in [Winsborough92]. The first implementation and evaluation in [PueblaH95].
- The drawbacks of PE for propagation are identified in [LeuschelS96] and a study of advantages of the integration presented in [Jones97].
- A first integration of AI and PE can be found in [PueblaHG97]
- An alternative formulation can be found in [Leuschel98]. Uses the notions of *Abstract Unfolding* and *Abstract resolution*.
- A general framework for the integration of abstract interpretation and program transformation can be found in [Cousot02].

Ongoing and Future Work

- Use Abstract Specialization for compile-time assertion checking.
- Use Abstract Specialization as a front end for an optimizing compiler in the Ciao system.
- Further explore the application of Abstract Specialization for Partial Evaluation tasks.
- Use our abstract interpreters for performing Binding Time Analysis for off-line partial evaluation.

The ASAP Project

- The ASAP IST-2001-38059 FET project: *Advanced Specialization and Analysis for Pervasive Computing*.
- In collaboration with Southampton and Bristol (UK) and Roskilde (DK).
- Basic idea: use automated techniques for reusing existing code in order to deploy it in resource-bounded systems.
- Program specialization has to be “resource-aware”.
- The cost-benefit relation of optimizations has to be taken into account.
- First result: An integrated system with the most advanced specialization and analysis techniques in the (C)LP community.

Comparison with On-Line Partial Evaluation

Feature	PE	AS
Information captured	concrete	abstract
Propagation of information	unfolding	abstract interpretation
Polyvariance control	global control	abstract domain
Guaranteeing termination	generalization	widening
Version selection	renaming	renaming
Materializing optimizations	unfolding	abstract executability
Fixpoint computations	no	yes
Propagation among atoms	no	yes
Infinite failure	non observable	observable
Low level optimization	non-applicable	applicable
Minimization of versions	no/yes	yes

- $|\text{AS (with unfolding)}|_{\text{concrete domain}} \geq \text{PE}$

Conclusions

- Abstract Specialization is generic. Can be used for many different kinds of information and optimization.
- Propagation of information is independent from performing the optimizations
- Not restricted to source to source optimizations:
 - ◇ abstract specialization generates the expanded program
 - ◇ which can then be combined with a low level optimizer
- It allows achieving an appropriate level of polyvariance
 - ◇ minimization allows reducing unnecessary polyvariance much
 - ◇ this allows “tentative” polyvariance which can be backtracked.
- A fully fledged integration of traditional partial evaluation and abstract specialization is most promising.

Bibliography on Abstract Specialization

- [1] F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López-García, and G. Puebla. The Ciao Prolog System. Reference Manual. The Ciao System Documentation Series–TR CLIP3/97.1, School of Computer Science, Technical University of Madrid (UPM), August 1997. System and on-line version of the manual available at <http://clip.dia.fi.upm.es/Software/Ciao/>.
- [2] 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.
- [3] 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.
- [4] 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. Truszczyński, and D. S. Warren, editors, *The Logic Programming Paradigm: a 25–Year Perspective*, pages 161–192. Springer-Verlag, July 1999.
- [5] M. Hermenegildo, G. Puebla, F. Bueno, and P. López-García. Abstract Verification and Debugging of Constraint Logic Programs. In *Recent Advances in Constraints*, number 2627 in LNCS, pages 1–14. Springer-Verlag, January 2003.
- [6] 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.
- [7] M. Hermenegildo, G. Puebla, K. Marriott, and P. Stuckey. Incremental Analysis of Logic Programs. In *International Conference on Logic Programming*, pages 797–811. MIT Press, June 1995.
- [8] 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.

- [9] 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.
- [10] 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.
- [11] G. Puebla, M. García de la Banda, K. Marriott, and P. Stuckey. Optimization of Logic Programs with Dynamic Scheduling. In *1997 International Conference on Logic Programming*, pages 93–107, Cambridge, MA, June 1997. MIT Press.
- [12] G. Puebla, J. Gallagher, and M. Hermenegildo. Towards Integrating Partial Evaluation in a Specialization Framework based on Generic Abstract Interpretation. In M. Leuschel, editor, *Proceedings of the ILPS'97 Workshop on Specialization of Declarative Programs*, October 1997. Post ILPS'97 Workshop.
- [13] G. Puebla and M. Hermenegildo. Implementation of Multiple Specialization in Logic Programs. In *Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics Based Program Manipulation*, pages 77–87. ACM Press, June 1995.
- [14] G. Puebla and M. Hermenegildo. Automatic Optimization of Dynamic Scheduling in Logic Programs. In *Programming Languages: Implementation, Logics, and Programs*, number 1140 in LNCS, Aachen, Germany, September 1996. Springer-Verlag. Poster abstract.
- [15] 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.
- [16] 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.

- [17] 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.
- [18] 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.
- [19] 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.

Downloading the systems

- Downloading `ciao`, `ciaopp`, `lpdoc`, and other CLIP software:
 - ◇ Standard distributions:
`http://www.clip.dia.fi.upm.es/Software`
 - ◇ Betas (in testing or completing documentation – ask webmaster for info):
`http://www.clip.dia.fi.upm.es/Software/Beta`
 - ◇ User's mailing list:
`ciao-users@clip.dia.fi.upm.es`