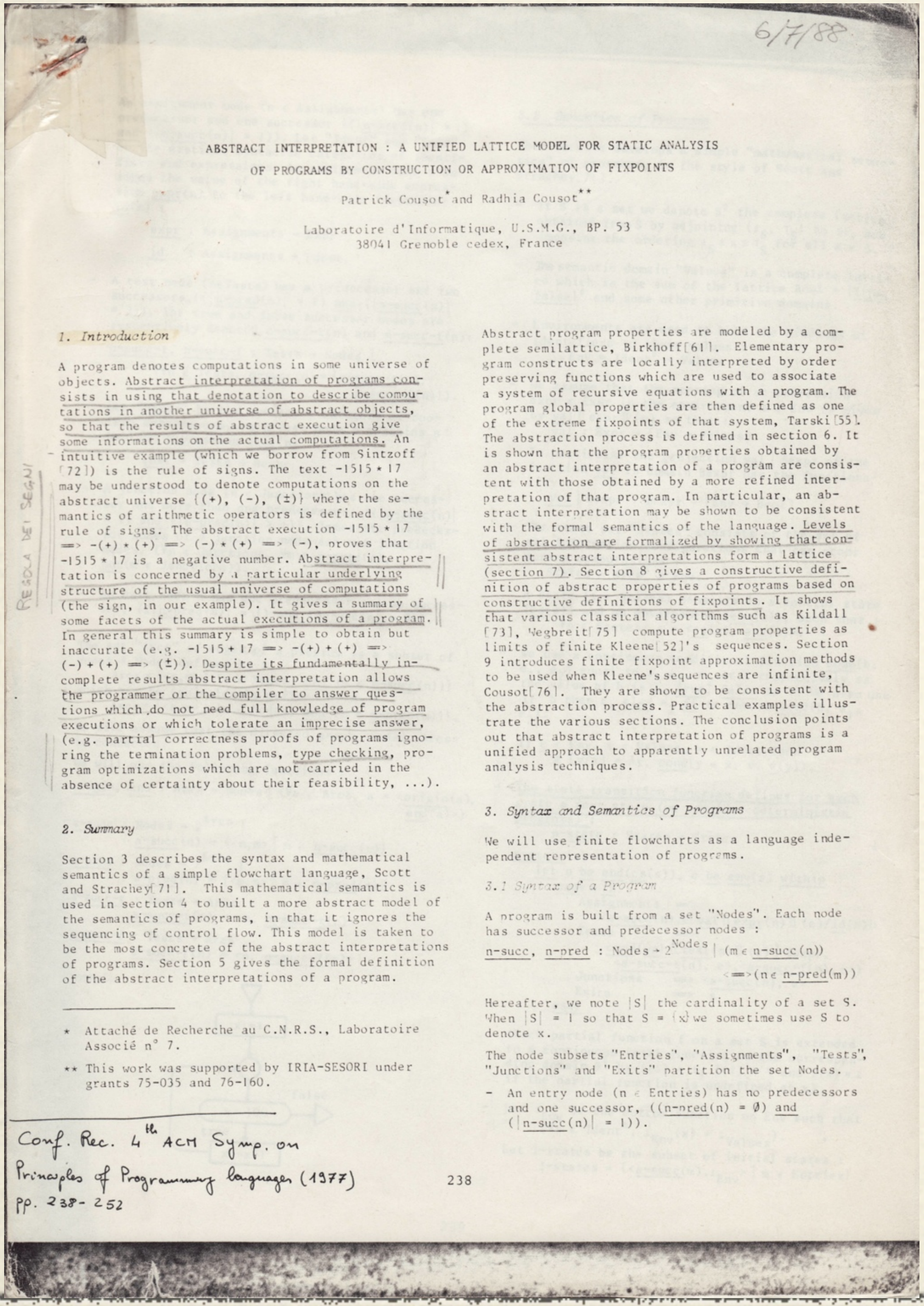
Abstract Interpretation in Pisa

From 1987 -:

The roots (*and my roots*)

- November 1987: *I was looking for a subject....*
- Giorgio gave me 2 papers:
  - *Abstract Interpretation or Partial Evaluation?*

The alternative....



6/7/88

ABSTRACT INTERPRETATION : A UNIFIED LATTICE MODEL FOR STATIC ANALYSIS
OF PROGRAMS BY CONSTRUCTION OR APPROXIMATION OF FIXPOINTS

Patrick Cousot* and Radhia Cousot**

Laboratoire d'Informatique, U.S.M.G., BP. 53
38041 Grenoble cedex, France

1. Introduction

A program denotes computations in some universe of objects. Abstract interpretation of programs consists in using that denotation to describe computations in another universe of abstract objects, so that the results of abstract execution give some informations on the actual computations. An intuitive example (which we borrow from Sintzoff [72]) is the rule of signs. The text $-1515 * 17$ may be understood to denote computations on the abstract universe $\{(+), (-), (\pm)\}$ where the semantics of arithmetic operators is defined by the rule of signs. The abstract execution $-1515 * 17 \Longrightarrow -(+) * (+) \Longrightarrow (-) * (+) \Longrightarrow (-)$, proves that $-1515 * 17$ is a negative number. Abstract interpretation is concerned by a particular underlying structure of the usual universe of computations (the sign, in our example). It gives a summary of some facets of the actual executions of a program. In general this summary is simple to obtain but inaccurate (e.g. $-1515 + 17 \Longrightarrow -(+) + (+) \Longrightarrow (-) + (+) \Longrightarrow (\pm)$). Despite its fundamentally incomplete results abstract interpretation allows the programmer or the compiler to answer questions which do not need full knowledge of program executions or which tolerate an imprecise answer, (e.g. partial correctness proofs of programs ignoring the termination problems, type checking, program optimizations which are not carried in the absence of certainty about their feasibility, ...).

REGOLA DEI SEGNI

2. Summary

Section 3 describes the syntax and mathematical semantics of a simple flowchart language, Scott and Strachey[71]. This mathematical semantics is used in section 4 to built a more abstract model of the semantics of programs, in that it ignores the sequencing of control flow. This model is taken to be the most concrete of the abstract interpretations of programs. Section 5 gives the formal definition of the abstract interpretations of a program. Abstract program properties are modeled by a complete semilattice, Birkhoff[61]. Elementary program constructs are locally interpreted by order preserving functions which are used to associate a system of recursive equations with a program. The program global properties are then defined as one of the extreme fixpoints of that system, Tarski[55]. The abstraction process is defined in section 6. It is shown that the program properties obtained by an abstract interpretation of a program are consistent with those obtained by a more refined interpretation of that program. In particular, an abstract interpretation may be shown to be consistent with the formal semantics of the language. Levels of abstraction are formalized by showing that consistent abstract interpretations form a lattice (section 7). Section 8 gives a constructive definition of abstract properties of programs based on constructive definitions of fixpoints. It shows that various classical algorithms such as Kildall [73], Wegbreit[75] compute program properties as limits of finite Kleene[52]'s sequences. Section 9 introduces finite fixpoint approximation methods to be used when Kleene's sequences are infinite, Cousot[76]. They are shown to be consistent with the abstraction process. Practical examples illustrate the various sections. The conclusion points out that abstract interpretation of programs is a unified approach to apparently unrelated program analysis techniques.

3. Syntax and Semantics of Programs

We will use finite flowcharts as a language independent representation of programs.

3.1 Syntax of a Program

A program is built from a set "Nodes". Each node has successor and predecessor nodes :

$$\underline{\text{n-succ}},\ \underline{\text{n-pred}} : \text{Nodes} \to 2^{\text{Nodes}} \mid (m \in \underline{\text{n-succ}}(n)) \Longleftrightarrow (n \in \underline{\text{n-pred}}(m))$$

Hereafter, we note $|S|$ the cardinality of a set S. When $|S| = 1$ so that $S = \{x\}$ we sometimes use S to denote x.

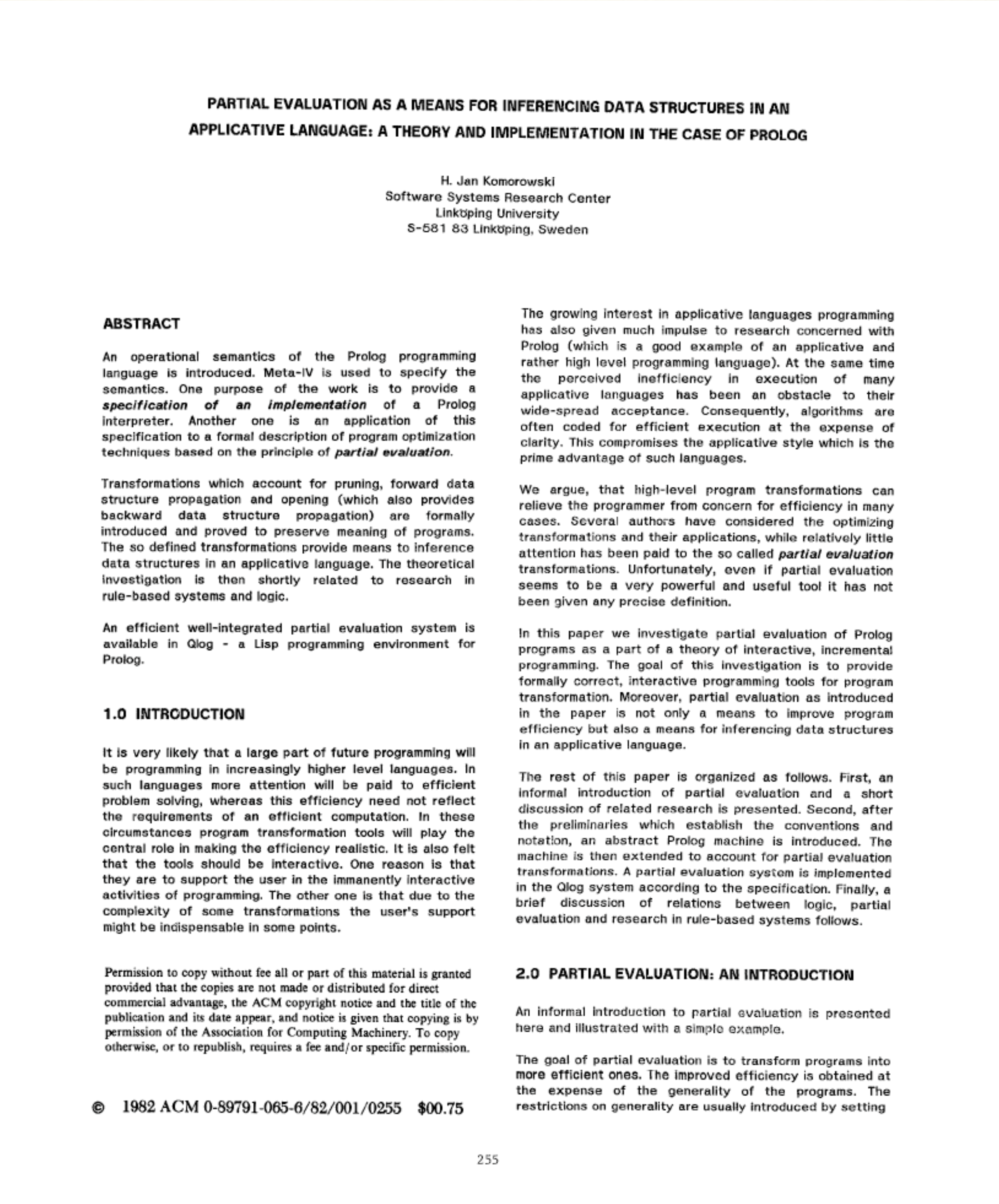
The node subsets "Entries", "Assignments", "Tests", "Junctions" and "Exits" partition the set Nodes.

- An entry node ($n \in$ Entries) has no predecessors and one successor, (($\underline{\text{n-pred}}(n) = \emptyset$) and ($|\underline{\text{n-succ}}(n)| = 1$)).

* Attaché de Recherche au C.N.R.S., Laboratoire Associé n° 7.

** This work was supported by IRIA-SESORI under grants 75-035 and 76-160.

Conf. Rec. 4th ACM Symp. on Principles of Programming languages (1977) pp. 238-252



PARTIAL EVALUATION AS A MEANS FOR INFERENCING DATA STRUCTURES IN AN APPLICATIVE LANGUAGE: A THEORY AND IMPLEMENTATION IN THE CASE OF PROLOG

H. Jan Komorowski
Software Systems Research Center
Linköping University
S-581 83 Linköping, Sweden

ABSTRACT

An operational semantics of the Prolog programming language is introduced. Meta-IV is used to specify the semantics. One purpose of the work is to provide a ***specification of an implementation*** of a Prolog interpreter. Another one is an application of this specification to a formal description of program optimization techniques based on the principle of ***partial evaluation***.

Transformations which account for pruning, forward data structure propagation and opening (which also provides backward data structure propagation) are formally introduced and proved to preserve meaning of programs. The so defined transformations provide means to inference data structures in an applicative language. The theoretical investigation is then shortly related to research in rule-based systems and logic.

An efficient well-integrated partial evaluation system is available in Qlog - a Lisp programming environment for Prolog.

1.0 INTRODUCTION

It is very likely that a large part of future programming will be programming in increasingly higher level languages. In such languages more attention will be paid to efficient problem solving, whereas this efficiency need not reflect the requirements of an efficient computation. In these circumstances program transformation tools will play the central role in making the efficiency realistic. It is also felt that the tools should be interactive. One reason is that they are to support the user in the immanently interactive activities of programming. The other one is that due to the complexity of some transformations the user's support might be indispensable in some points.

The growing interest in applicative languages programming has also given much impulse to research concerned with Prolog (which is a good example of an applicative and rather high level programming language). At the same time the perceived inefficiency in execution of many applicative languages has been an obstacle to their wide-spread acceptance. Consequently, algorithms are often coded for efficient execution at the expense of clarity. This compromises the applicative style which is the prime advantage of such languages.

We argue, that high-level program transformations can relieve the programmer from concern for efficiency in many cases. Several authors have considered the optimizing transformations and their applications, while relatively little attention has been paid to the so called ***partial evaluation*** transformations. Unfortunately, even if partial evaluation seems to be a very powerful and useful tool it has not been given any precise definition.

In this paper we investigate partial evaluation of Prolog programs as a part of a theory of interactive, incremental programming. The goal of this investigation is to provide formally correct, interactive programming tools for program transformation. Moreover, partial evaluation as introduced in the paper is not only a means to improve program efficiency but also a means for inferencing data structures in an applicative language.

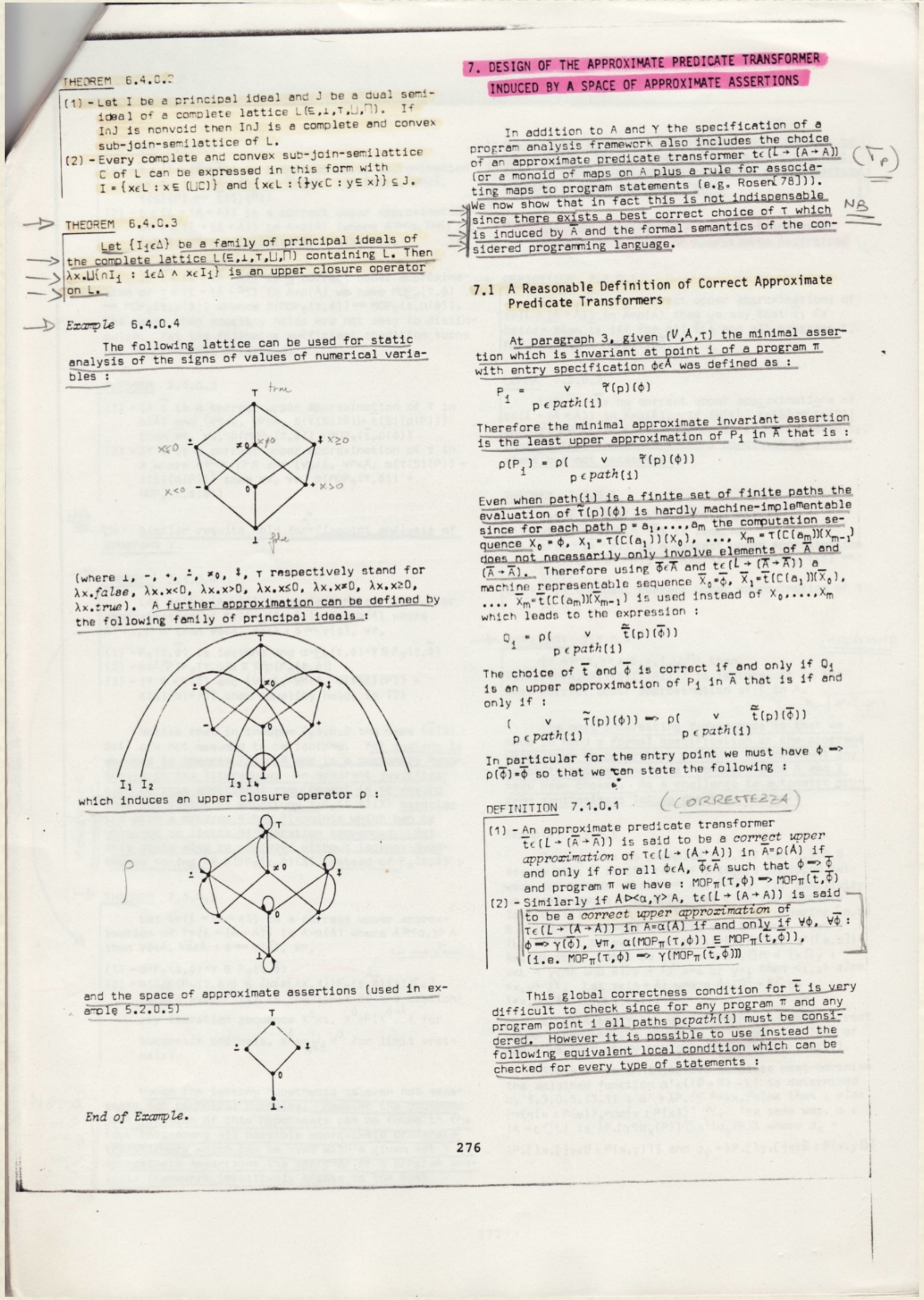
The rest of this paper is organized as follows. First, an informal introduction of partial evaluation and a short discussion of related research is presented. Second, after the preliminaries which establish the conventions and notation, an abstract Prolog machine is introduced. The machine is then extended to account for partial evaluation transformations. A partial evaluation system is implemented in the Qlog system according to the specification. Finally, a brief discussion of relations between logic, partial evaluation and research in rule-based systems follows.

2.0 PARTIAL EVALUATION: AN INTRODUCTION

An informal introduction to partial evaluation is presented here and illustrated with a simple example.

The goal of partial evaluation is to transform programs into more efficient ones. The improved efficiency is obtained at the expense of the generality of the programs. The restrictions on generality are usually introduced by setting

The choice....



THEOREM 6.4.0.2

(1) - Let I be a principal ideal and J be a dual semi-ideal of a complete lattice $L(\sqsubseteq,\bot,\top,\sqcup,\sqcap)$. If $I\cap J$ is nonvoid then $I\cap J$ is a complete and convex sub-join-semilattice of L.

(2) - Every complete and convex sub-join-semilattice C of L can be expressed in this form with $I=\{x\in L : x\sqsubseteq(\sqcup C)\}$ and $\{x\in L : \{\exists y\in C : y\sqsubseteq x\}\}\subseteq J$.

THEOREM 6.4.0.3

Let $\{I_i\in\Delta\}$ be a family of principal ideals of the complete lattice $L(\sqsubseteq,\bot,\top,\sqcup,\sqcap)$ containing L. Then $\lambda x.\sqcup\{\cap I_i : i\in\Delta \wedge x\in I_i\}$ is an upper closure operator on L.

Example 6.4.0.4

The following lattice can be used for static analysis of the signs of values of numerical variables :

(where $\bot$, $-$, $+$, $\dot{-}$, $\neq 0$, $\dot{+}$, $\top$ respectively stand for $\lambda x.\mathit{false}$, $\lambda x.x<0$, $\lambda x.x>0$, $\lambda x.x\leq 0$, $\lambda x.x\neq 0$, $\lambda x.x\geq 0$, $\lambda x.\mathit{true}$). A further approximation can be defined by the following family of principal ideals :

which induces an upper closure operator ρ :

and the space of approximate assertions (used in example 5.2.0.5)

End of Example.

7. DESIGN OF THE APPROXIMATE PREDICATE TRANSFORMER INDUCED BY A SPACE OF APPROXIMATE ASSERTIONS

In addition to A and Υ the specification of a program analysis framework also includes the choice of an approximate predicate transformer $t\in(L\to(A\to A))$ (or a monoid of maps on A plus a rule for associating maps to program statements (e.g. Rosen[78])). We now show that in fact this is not indispensable since there exists a best correct choice of τ which is induced by $\bar{A}$ and the formal semantics of the considered programming language. $(\overline{\tau}_\rho)$ NB

7.1 A Reasonable Definition of Correct Approximate Predicate Transformers

At paragraph 3, given (V,A,τ) the minimal assertion which is invariant at point i of a program π with entry specification $\phi\in A$ was defined as :

$$P_i = \bigvee_{p\in path(i)} \tilde{\tau}(p)(\phi)$$

Therefore the minimal approximate invariant assertion is the least upper approximation of P_i in $\bar{A}$ that is :

$$\rho(P_i) = \rho(\bigvee_{p\in path(i)} \tilde{\tau}(p)(\phi))$$

Even when path(i) is a finite set of finite paths the evaluation of $\tilde{\tau}(p)(\phi)$ is hardly machine-implementable since for each path $p=a_1,\ldots,a_m$ the computation sequence $X_0=\phi$, $X_1=\tau(C(a_1))(X_0)$, ..., $X_m=\tau(C(a_m))(X_{m-1})$ does not necessarily only involve elements of $\bar{A}$ and $(\bar{A}\to\bar{A})$. Therefore using $\bar{\Phi}\in\bar{A}$ and $t\in(L\to(\bar{A}\to\bar{A}))$ a machine representable sequence $\bar{X}_0=\bar{\phi}$, $\bar{X}_1=\bar{t}(C(a_1))(\bar{X}_0)$, ... $\bar{X}_m=\bar{t}(C(a_m))(\bar{X}_{m-1})$ is used instead of $X_0,\ldots,X_m$ which leads to the expression :

$$Q_i = \rho(\bigvee_{p\in path(i)} \tilde{\bar{t}}(p)(\bar{\phi}))$$

The choice of $\bar{t}$ and $\bar{\phi}$ is correct if and only if Q_i is an upper approximation of P_i in $\bar{A}$ that is if and only if :

$$(\bigvee_{p\in path(i)} \tilde{\tau}(p)(\phi)) \Rightarrow \rho(\bigvee_{p\in path(i)} \tilde{\bar{t}}(p)(\bar{\phi}))$$

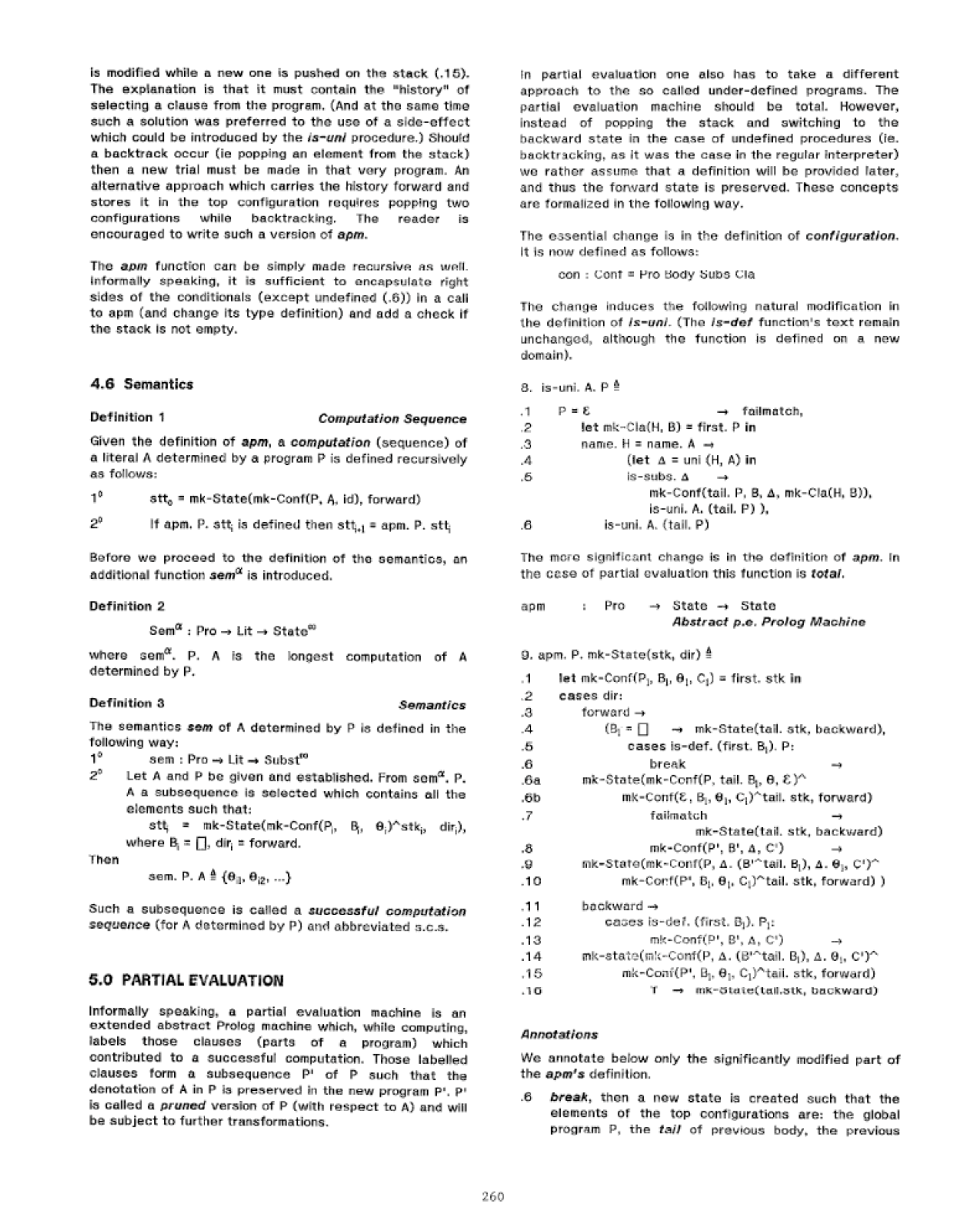
In particular for the entry point we must have $\phi\Rightarrow\rho(\bar{\phi})=\bar{\phi}$ so that we can state the following :

DEFINITION 7.1.0.1 (CORRESTEZZA)

(1) - An approximate predicate transformer $\bar{t}\in(L\to(\bar{A}\to\bar{A}))$ is said to be a *correct upper approximation* of $\tau\in(L\to(A\to A))$ in $\bar{A}=\rho(A)$ if and only if for all $\phi\in A$, $\bar{\Phi}\in\bar{A}$ such that $\phi\Rightarrow\bar{\Phi}$ and program π we have : $MOP_\pi(\tau,\phi)\Rightarrow MOP_\pi(\bar{t},\bar{\Phi})$

(2) - Similarly if $A\triangleright\langle\alpha,\gamma\rangle A$, $t\in(L\to(A\to A))$ is said to be a *correct upper approximation* of $\tau\in(L\to(A\to A))$ in $A=\alpha(A)$ if and only if $\forall\phi$, $\forall\bar{\phi}$: $\phi\Rightarrow\gamma(\bar{\phi})$, $\forall\pi$, $\alpha(MOP_\pi(\tau,\phi))\sqsubseteq MOP_\pi(t,\bar{\phi}))$, (i.e. $MOP_\pi(\tau,\phi)\Rightarrow\gamma(MOP_\pi(\bar{t},\bar{\phi}))$)

This global correctness condition for $\bar{t}$ is very difficult to check since for any program π and any program point i all paths $p\in path(i)$ must be considered. However it is possible to use instead the following equivalent local condition which can be checked for every type of statements :



is modified while a new one is pushed on the stack (.15). The explanation is that it must contain the "history" of selecting a clause from the program. (And at the same time such a solution was preferred to the use of a side-effect which could be introduced by the *is-uni* procedure.) Should a backtrack occur (ie popping an element from the stack) then a new trial must be made in that very program. An alternative approach which carries the history forward and stores it in the top configuration requires popping two configurations while backtracking. The reader is encouraged to write such a version of ***apm***.

The ***apm*** function can be simply made recursive as well. Informally speaking, it is sufficient to encapsulate right sides of the conditionals (except undefined (.6)) in a call to apm (and change its type definition) and add a check if the stack is not empty.

4.6 Semantics

Definition 1 ***Computation Sequence***

Given the definition of ***apm***, a ***computation*** (sequence) of a literal A determined by a program P is defined recursively as follows:

1^0 stt_0 = mk-State(mk-Conf(P, A, id), forward)

2^0 If apm. P. stt_i is defined then stt_{i+1} = apm. P. stt_i

Before we proceed to the definition of the semantics, an additional function ***sem***$^\alpha$ is introduced.

Definition 2

Sem^α : Pro → Lit → $State^\infty$

where sem^α. P. A is the longest computation of A determined by P.

Definition 3 ***Semantics***

The semantics ***sem*** of A determined by P is defined in the following way:

1^0 sem : Pro → Lit → $Subst^\infty$

2^0 Let A and P be given and established. From sem^α. P. A a subsequence is selected which contains all the elements such that:
stt_i = mk-State(mk-Conf(P_i, B_i, θ_i)^stk_i, dir_i), where B_i = □, dir_i = forward.

Then

sem. P. A ≜ {θ_{i1}, θ_{i2}, ...}

Such a subsequence is called a ***successful computation sequence*** (for A determined by P) and abbreviated s.c.s.

5.0 PARTIAL EVALUATION

Informally speaking, a partial evaluation machine is an extended abstract Prolog machine which, while computing, labels those clauses (parts of a program) which contributed to a successful computation. Those labelled clauses form a subsequence P' of P such that the denotation of A in P is preserved in the new program P'. P' is called a ***pruned*** version of P (with respect to A) and will be subject to further transformations.

In partial evaluation one also has to take a different approach to the so called under-defined programs. The partial evaluation machine should be total. However, instead of popping the stack and switching to the backward state in the case of undefined procedures (ie. backtracking, as it was the case in the regular interpreter) we rather assume that a definition will be provided later, and thus the forward state is preserved. These concepts are formalized in the following way.

The essential change is in the definition of ***configuration***. It is now defined as follows:

con : Conf = Pro Body Subs Cla

The change induces the following natural modification in the definition of ***is-uni***. (The ***is-def*** function's text remain unchanged, although the function is defined on a new domain).

```
8.  is-uni. A. P ≜

.1      P = ε                            →  failmatch,
.2          let mk-Cla(H, B) = first. P in
.3          name. H = name. A  →
.4                  (let  Δ = uni (H, A) in
.5                  is-subs. Δ      →
.6                      mk-Conf(tail. P, B, Δ, mk-Cla(H, B)),
.7                      is-uni. A. (tail. P) ),
.6              is-uni. A. (tail. P)
```

The more significant change is in the definition of ***apm***. In the case of partial evaluation this function is ***total***.

apm : Pro → State → State
Abstract p.e. Prolog Machine

```
9. apm. P. mk-State(stk, dir) ≜

.1      let mk-Conf(P1, B1, θ1, C1) = first. stk in
.2      cases dir:
.3          forward →
.4              (B1 = □     →  mk-State(tail. stk, backward),
.5                  cases is-def. (first. B1). P:
.6                      break                           →
.6a         mk-State(mk-Conf(P, tail. B1, θ, ε)^
.6b                 mk-Conf(ε, B1, θ1, C1)^tail. stk, forward)
.7                      failmatch                       →
                                mk-State(tail. stk, backward)
.8                      mk-Conf(P', B', Δ, C')          →
.9          mk-State(mk-Conf(P, Δ. (B'^tail. B1), Δ. θ1, C')^
.10                 mk-Conf(P', B1, θ1, C1)^tail. stk, forward) )

.11         backward →
.12             cases is-def. (first. B1). P1:
.13                     mk-Conf(P', B', Δ, C')          →
.14         mk-state(mk-Conf(P, Δ. (B'^tail. B1), Δ. θ1, C')^
.15                 mk-Conf(P', B1, θ1, C1)^tail. stk, forward)
.16                     T   →  mk-State(tail.stk, backward)
```

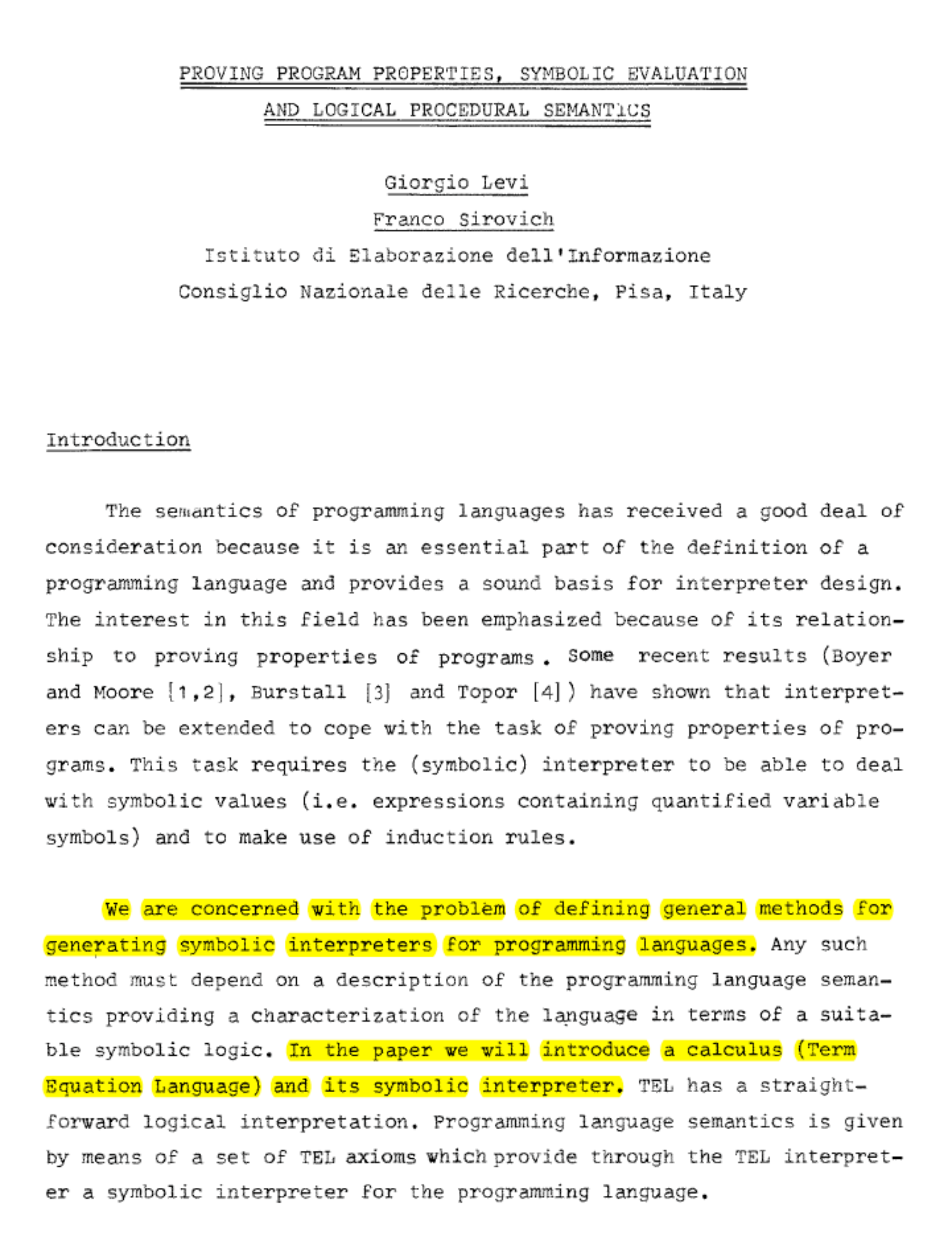
Annotations

We annotate below only the significantly modified part of the ***apm's*** definition.

.6 ***break***, then a new state is created such that the elements of the top configurations are: the global program P, the ***tail*** of previous body, the previous

The beginning.....

- Giorgio was of course already interested in semantics and correctness of *symbolic interpreters*!!!
  - *Giorgio Levi, Franco Sirovich: Proving Program Properties, Symbolic Evaluation and Logical Procedural Semantics. MFCS 1975: 294-301*



PROVING PROGRAM PROPERTIES, SYMBOLIC EVALUATION AND LOGICAL PROCEDURAL SEMANTICS

Giorgio Levi
Franco Sirovich
Istituto di Elaborazione dell'Informazione
Consiglio Nazionale delle Ricerche, Pisa, Italy

Introduction

The semantics of programming languages has received a good deal of consideration because it is an essential part of the definition of a programming language and provides a sound basis for interpreter design. The interest in this field has been emphasized because of its relationship to proving properties of programs. Some recent results (Boyer and Moore [1,2], Burstall [3] and Topor [4]) have shown that interpreters can be extended to cope with the task of proving properties of programs. This task requires the (symbolic) interpreter to be able to deal with symbolic values (i.e. expressions containing quantified variable symbols) and to make use of induction rules.

We are concerned with the problem of defining general methods for generating symbolic interpreters for programming languages. Any such method must depend on a description of the programming language semantics providing a characterization of the language in terms of a suitable symbolic logic. In the paper we will introduce a calculus (Term Equation Language) and its symbolic interpreter. TEL has a straightforward logical interpretation. Programming language semantics is given by means of a set of TEL axioms which provide through the TEL interpreter a symbolic interpreter for the programming language.

...and Barbuti?

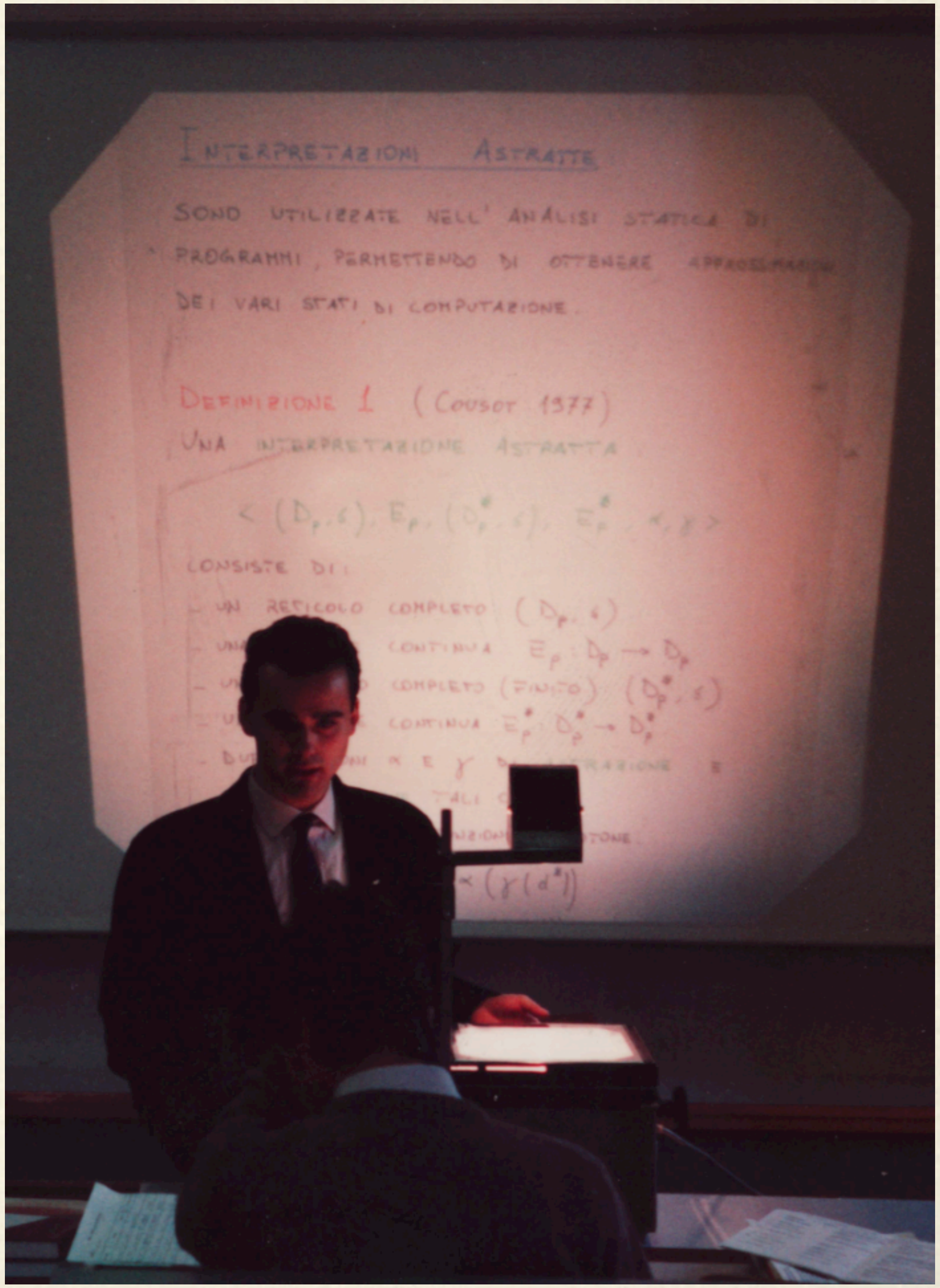
- A great paper*: Roberto Barbuti, Alberto Martelli: A Structured Approach to Static Semantics Correctness. Sci. Comput. Program. 3(3): 279-311 (1983).
  - *This is where I have understood correctness of static semantics!*
  - *.....simple and clean!*

The beginning...

- In Pisa: *Roberto Barbuti, Roberto Giacobazzi, Giorgio Levi: A General Framework for Semantics-Based Bottom-Up Abstract Interpretation of Logic Programs. ITCS 1989 and later in ACM TOPLAS.*

- In Padova: *Christian Codognet, Philippe Codognet, Gilberto Filé: Yet Another Intelligent Backtracking Method. ICLP/SLP 1988: 447-465.*

- The big (theological) deal: **bottom-up** or **top-down?**



INTERPRETAZIONI ASTRATTE

SONO UTILIZZATE NELL' ANALISI STATICA DI PROGRAMMI, PERMETTENDO DI OTTENERE APPROSSIMAZIONI DEI VARI STATI DI COMPUTAZIONE.

DEFINIZIONE 1 (COUSOT 1977)

UNA INTERPRETAZIONE ASTRATTA

$< (D_p, \leq), E_p, (D_p^\#, \leq), E_p^\#, \alpha, \gamma >$

CONSISTE DI:

- UN RETICOLO COMPLETO $(D_p, \leq)$
- UNA [illegible] CONTINUA $E_p : D_p \rightarrow D_p$
- U[illegible] COMPLETO (FINITO) $(D_p^\#, \leq)$
- U[illegible] CONTINUA $E_p^\# : D_p^\# \rightarrow D_p^\#$
- DU[illegible]NI α E γ D[illegible]RAZIONE E

[illegible] TALI C[illegible]

[illegible]NZIONE [illegible]ZIONE.

[illegible] $\alpha(\gamma(d^\#))$

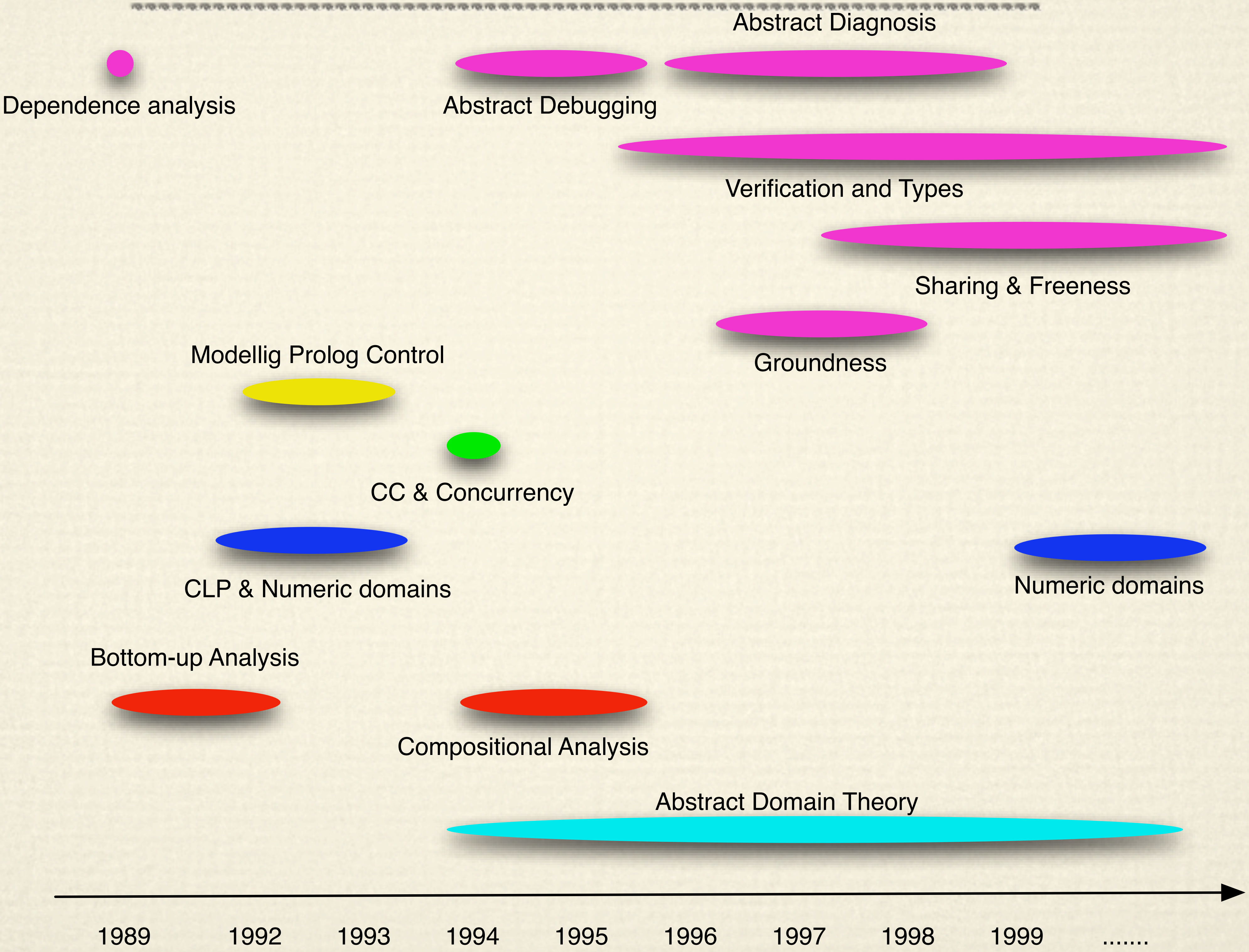
The great days!!

- In Pisa in the 90s:
  - The best group in semantics of Logic Programming!
  - Strong mathematical bases!
  - Challenging projects!
  - International environment
  - Enthusiasm!

The Cousots exist!

ICLP'91: a joint workshop on
<u>Abstract Interpretation in Logic Programming</u>
*... in **Paris**!*
and then: WSA, SAS, etc..

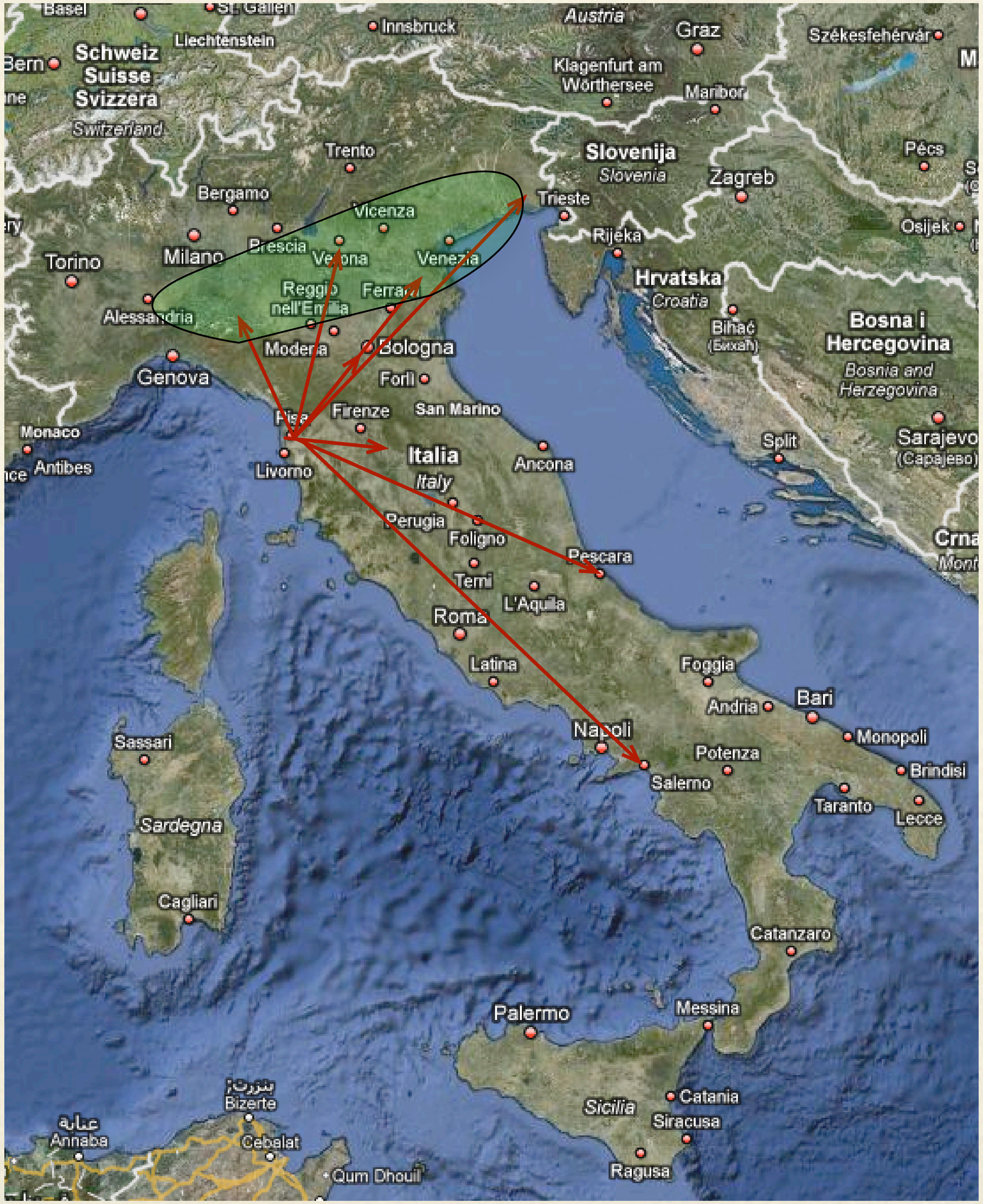
The Abstract Interpretation *slice*



Abstract Interpretation *and People*

Giorgio Levi
Roberto Barbuti
Roberto Giacobazzi
Michael Codish
Michael Maher
Roberto Bagnara
Saumya Debray
Enea Zaffanella
Francesco Ranzato
Francesca Scozzari
Agostino Cortesi
Catuscia Palamidessi
Moreno Falaschi
Gilberto Filè
Giuliana Vitiello
Maria Chiara Meo
Marco Comini
Paolo Volpe
Roberta Gori
Fausto Spoto
Gianluca Amato
Francesca Levi
Sergio Maffeis
Patricia M. Hill
Francois Fages

Spreading ideas *and people*



The chance of being in Pisa