FIXED POINT APPROACH TO THE APPROXIMATE SEMANTIC ANALYSIS OF PROGRAMS

Patrick Cousot* and Radhia Cousot**
Université Scientifique et Médicale de Grenoble
(June 1977)

Parts of sections 2 and 4 are based on a paper presented at the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, Calif., January 1977 [13], section 7 is based on a paper presented at the ACM Conference on Language Design for Reliable Software, Raleigh, North-Carolina, March 1977 [14], and parts of sections 2 and 6 are based on a paper presented at the SIGACT-SIGPLAN Symposium on Artificial Intelligence & Programming Languages, Rochester, New-York, August 1977 [16].

This work was supported in part by (*) CNRS, Laboratoire Associé n°7 and ATP D3119 and in part by (**) IRIA under grant SESORI-76160.

Authors'address : Mathématiques Appliquées Informatique, Laboratoire Associé au CNRS n°7, Université Scientifique et Médicale, B.P.53, 38041 Grenoble-Cedex, France.

Abstract : Abstract interpretation constitutes a mathematical model for static analysis of programs. The information to be gathered about programs are modeled in a complete lattice. According to the semantics of the utilized language, elementary instructions can be interpreted by order-preserving functions. This permits a system of recursive equations to be associated with any particular program. The determination of properties of that program then consists in solving the corresponding fixed point equations. For continuous equations the exact solution can be constructed iteratively by Jacobi's successive approximations, but in practice any chaotic iteration method is shown to fit.

Standard arguments on decidability show that convergence of successive iterates may happen not to be guaranteed. Mathematical techniques are reviewed to cope with the determination of undecidable properties of programs. In particular, we introduce structural and more generally computational approximation methods which can always be used in practice to mechanically discover a correct approximation of the exact but unreachable solution to the equations.

Abstract interpretation of programs provides a unified approach to apparently unrelated program analysis techniques. It is shown to be powerful enough to tackle with theoretical as well as practical problems such as denotational semantics of programs, proofs of partial correctness, proofs of termination, symbolic execution, analysis of program performance, type verification or discovery, global data flow analysis, finite or infinite state program analysis.

Key Words and Phrases : abstract interpretations, static analysis, lattice, system of equations, fixed points construction, fixed points approximation, chaotic iterative methods, program semantics, logical analysis, correctness, termination, symbolic execution, performance analysis, type discovery, finite or infinite state program analysis, global data flow analysis.

CR Categories : 3.66, 4.0, 4.12, 4.13, 4.2, 4.42, 5.24, 5.30, 5.7

1. INTRODUCTION

In recent years considerable effort has been devoted to the development of rational techniques for conducting analysis of programs. The general term *analysis of programs* covers a wide variety of domains and specific techniques. *Semantic analysis* attempts to specify the "meaning" of a program, (see e.g. [54]), *logical analysis* is used either to verify the program with respect to a specification or to prove that the program contains an error (see e.g. [32]), *compile time analysis* includes techniques such as type verification or type discovery which are used to check weak properties of programs (see e.g.[14]) or global data flow analysis preceeding program optimization (see e.g.[62]), *performance analysis* tries to anticipate the running time of programs (see e.g.[64]).

In spite of the apparent diversity of these multifarious domains we have had for a long time the feeling that very similar techniques were used and thought that a common model would high-light the underlying unity and provides some insight in each of these specialized and somewhat artificially disjoint research areas.

This paper deals with *abstract interpretation* a mathematical model for static analysis of programs. The information to be gathered about programs are modeled in a complete lattice. According to the semantics of the utilized language elementary instructions can be interpreted by order-preserving functions. This permits a system of recursive equations to be associated with any particular program. The determination of properties of that program then consists in solving the corresponding fixed point equations. Under continuity hypothesis the exact solution can be constructed iteratively by Jacobi's successive approximations but this result is generalized and we show that in practice any chaotic iteration method would fit. Standard arguments on decidability show that the iterates may happen not to converge toward the desired solution in a finite number of steps. Therefore mathematical techniques must be reviewed to cope with the determination of undecidable properties of programs. The classical methods are formal resolution of the equations and utilization of mathematical induction. However this

cannot lead to completely mechanized techniques [illegible] and more gene-
rally computational approximation methods which [illegible] in practice for an automatic
discovery of a correct approximation of the exact [illegible] solution to the equations.

Clearly, parts of this fixed point approach to [illegible] programs have been explicitly
or implicitly suggested in the litterature (for example [12],[17],[28],[30],[illegible],[44],[illegible],[illegible],
[61], [62], [65], [66]). However the originality of our work was to synthesize the various ap-
proaches and to enlarge them in order to be able to tackle with theoretic as well as very prac-
tical problems.

In Section 2 we introduce the mathematical model used in abstract interpretation of pro-
grams. It is illustrated by a very simple and intuitive example. Section 3 applies the model to
performance analysis of programs, the intention is to render the connection with numerical anal-
ysis straightforward. Section 4 copes with the practical problem of discovering undecidable
properties of programs. The remaining sections point out the wide range of applicability of the
model. We describe the application of abstract interpretation of programs to mathematical se-
mantics (Section 5), logical analysis of programs (Section 6), compile-time analysis (Section
7). Implication of this work and the necessary further researches are discussed in the conclu-
sion (Section 8).

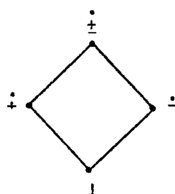
2. THE MATHEMATICAL MODEL USED IN ABSTRACT INTERPRETATION OF PROGRAMS

2.1 THE COMPLETE LATTICE OF PROPERTIES

Example : Let us introduce the *abstract interpretation* of programs by means of a very intui-
tive and trivial example. Suppose we are interested in discovering the sign of the integer va-
riable x in the (non-terminating) program :

```
x:=1; while true do x:=x+1 od;
```

Roughly speaking, program analysis requires the determination for each program point i of an
invariant property P_i known to hold each time control reaches i during execution, independently
of the path taken to reach the program point i. Let us introduce notations for these properties :
we denote by $\dot{+}$ the fact that x is positive, by $\dot{-}$ that x is negative, and by $\dot{\pm}$ the fact that x
is an integer whose sign is unknown. We denote by $\bot$ the fact that x is not initialized. The
set $L = \{\bot, \dot{+}, \dot{-}, \dot{\pm}\}$ of properties is ordered by the partial ordering relation $\sqsubseteq$ defined by the
following Hasse diagram :



For instance $\dot{+} \sqsubseteq \dot{\pm}$ since the assertion that x is a positive integer is less unprecise than the
assertion that x is simply an integer. *End of Example.*

The set L of properties is a *complete semilattice* with *partial ordering* $\sqsubseteq$ and *least upper
bound* or *join* of two elements $\sqcup$. We also assume that L has an *infimum* denoted $\bot$. (The defini-
tions and mathematical properties of these notions can be found in many places, for example [5]).

Since L is a join-semilattice with infimum, it is a *complete lattice* ([5],Ch.V,§3). We will respectively denote by $\coprod$, $\sqcap$, $\prod$ the join of a set of elements, *greatest lower bound* (or *meet*) of two elements, and meet of a set of elements of L. Since L is a complete lattice it has a *supremum* denoted by $\top$.

We will use the fact that L is a complete lattice in the formal reasoning whereas for the practical applications we will need only to implement the operations $\sqcup$ and $\sqsubseteq$.

Example : The partial ordering $\sqsubseteq$ has been defined by : $\bot \sqsubseteq \bot \sqsubseteq \dot{+} \sqsubseteq \dot{+} \sqsubseteq \dot{\pm} \sqsubseteq \dot{\pm}$ and $\bot \sqsubseteq \dot{-} \sqsubseteq \dot{-} \sqsubseteq \dot{\pm}$ where $\dot{\pm}$ is the supremum otherwise denoted by $\top$. This implies that the join of two properties is defined by $\dot{+} \sqcup \dot{+} = \dot{+}$, $\dot{+} \sqcup \dot{-} = \dot{\pm}$, $\dot{+} \sqcup \dot{\pm} = \dot{\pm}$, $\dot{+} \sqcup \bot = \dot{+}$, etc. *End of Example.*

2.2 SYSTEM OF EQUATIONS ASSOCIATED WITH A PROGRAM

Example : Let us further associate invariants P_1, P_2, P_3 with various points {1}, {2} and {3} of the program :

x:=1 {1}; <u>while</u> <u>true</u> <u>do</u> {2} x:=x+1 {3} <u>od</u>;

The value of P_1, P_2, P_3 may be $\dot{\pm}$, $\dot{+}$, $\dot{-}$ or $\bot$ depending on the dynamic properties of x at the respective program points {1}, {2} or {3}.

According to the semantics of usual programming languages we know that :

- x is positive at program point {1} since it is equal to 1. Therefore : $P_1 = \dot{+}$.
- the sign of x at point {2} may be P_1 when coming from point {1} or P_3 when coming from {3}. Therefore : $P_2 = P_1 \sqcup P_3$.
- finally suppose the sign of x is P_2, then the sign of x+1 is $P_2 \boxplus \dot{+}$, where the operator $\boxplus$ is defined by the rules of signs :

$$\begin{array}{l} \dot{+} \boxplus \dot{+} = \dot{+} \\ \dot{-} \boxplus \dot{+} = \dot{\pm} \\ \dot{\pm} \boxplus \dot{+} = \dot{\pm} \\ \bot \boxplus \dot{+} = \bot \\ \ldots \end{array}$$

Hence the sign P_3 of x after the assignment x:=x+1 is $P_2 \boxplus \dot{+}$. Therefore : $P_3 = P_2 \boxplus \dot{+}$.

Notice that our simple reasoning permits to establish a system of three relations between the three invariants P_1, P_2, P_3 :

$$\left\{ \begin{array}{l} P_1 = \dot{+} \\ P_2 = P_1 \sqcup P_3 \\ P_3 = P_2 \boxplus \dot{+} \end{array} \right.$$

Since P_1, P_2, P_3 need not satisfy any other constraint, any solution to the system of equations:

$$\left\{ \begin{array}{l} X_1 = \dot{+} \\ X_2 = X_1 \sqcup X_3 \\ X_3 = X_2 \boxplus \dot{+} \end{array} \right.$$

would be an acceptable candidate for the invariants. *End of Example.*

According to the semantics of the utilized programming language, the assertion P_i associated with program point {i} is a function f_i of the assertions $P_1, \ldots, P_n$ associated with the various points {1},..., {n} of the program. Therefore the desired properties $P_1, \ldots, P_n$ must be one of the solutions to a *system of mutually recursive equations* with n variables of the form :

$$\left\{ \begin{array}{l} X_1 = f_1(X_1, \ldots, X_n) \\ \ldots \\ X_n = f_n(X_1, \ldots, X_n) \end{array} \right.$$

abbreviated by a *fixed point equation* $X=F(X)$ where X is the vector $\langle X_1,\ldots,X_n\rangle$.

2.3 EXISTENCE OF A LEAST SOLUTION TO THE SYSTEM OF EQUATIONS

Example : Solutions to the system of equations :

$$\begin{cases} X_1 = \dot{+} & = f_1(X_1,X_2,X_3) \\ X_2 = X_1 \sqcup X_3 & = f_2(X_1,X_2,X_3) \\ X_3 = X_2 \boxplus \dot{+} & = f_3(X_1,X_2,X_3) \end{cases}$$

exist, and in general are not unique :

$$\text{(a)} \begin{bmatrix} X_1 = \dot{+} \\ X_2 = \dot{+} \\ X_3 = \dot{+} \end{bmatrix} \qquad \text{(b)} \begin{bmatrix} X_1 = \dot{+} \\ X_2 = \dot{\pm} \\ X_3 = \dot{\pm} \end{bmatrix}$$

However, a best solution (a) exists since $\dot{+}$ is a more precise result than $\dot{\pm}$. Since $\dot{+} \sqsubseteq \dot{\pm}$, (a) is the least solution to the system of equations. *End of Example.*

HYPOTHESIS 2.3.1 *The set* L *of properties is assumed to be a complete lattice* $(\sqsubseteq,\bot,\top,\sqcup,\sqcap,\bigsqcup,\bigsqcap)$.

LEMMA 2.3.2 *The set* L^n *is a complete lattice.*

The direct product L^n is the set of all sequences $\langle X_1,\ldots,X_n\rangle$ with n elements belonging to L. Since L is a complete lattice, it is easy to prove that the "componentwise" definition of $\sqsubseteq_n$, $\bot_n$, $\top_n$, $\sqcup_n$, $\sqcap_n$, $\bigsqcup_n$, $\bigsqcap_n$ make L^n a complete lattice ([5],Ch.V,§1). As usual we have :

$$\{\langle X_1,\ldots,X_n\rangle \sqsubseteq_n \langle Y_1,\ldots,Y_n\rangle\} \iff \{(X_i \sqsubseteq Y_i),\ \forall i\in[1,n]\}$$
$$\langle X_1,\ldots,X_n\rangle \sqcup_n \langle Y_1,\ldots,Y_n\rangle \quad = \quad \langle X_1 \sqcup Y_1,\ldots,X_n \sqcup Y_n\rangle$$

etc.

We drop the subscript n in the notations $\sqsubseteq_n$, $\bot_n$,... when unambiguously available from context.

DEFINITION 2.3.3 *A function* $\varphi : D \rightarrow D'$ *on the poset* $(D,\leq)$ *to the poset* $(D',\sqsubseteq)$ *is* order-preserving (*synonymously*, monotone *or* isotone) *if and only if :* $\{\forall (x,y)\in D^2,\ (x\leq y) \Longrightarrow (\varphi(x) \sqsubseteq \varphi(y))\}$

Note that when D and D' are join semi-lattices this definition is equivalent to :
$\{\forall (x,y)\in D^2,\ (\varphi(x) \sqcup \varphi(y)) \sqsubseteq \varphi(x \sqcup y)\}$

HYPOTHESIS 2.3.4 *The functions* f_i, $i\in[1,n]$ *are assumed to be monotone functions on the poset* $(L^n,\sqsubseteq_n)$ *to the poset* $(L,\sqsubseteq)$.

LEMMA 2.3.5 *The function* F *on the complete lattice* L^n *to itself is monotone.*

This is a direct consequence of the hypothesis that the f_i are monotone. The main consequence of the lemma 2.3.5 is that F has *fixed points*, that is there exists some $P\in L^n$ such that $P=F(P)$. In general the number of fixed points of F is infinite. Fortunately there exists a unique *least fixed point* P of F such that $P=F(P)$ and if $Q=F(Q)$ then $P \sqsubseteq_n Q$. The least fixed point P of F is chosen to be the solution to the system of equations $X=F(X)$.

THEOREM 2.3.6 *Any monotone map* φ *of a complete lattice* L $(\sqsubseteq,\bot,\top,\sqcup,\sqcap,\bigsqcup,\bigsqcap)$ *into itself has a least fixed point* $lfp(\varphi)$ *defined by :* $lfp(\varphi) = \bigsqcap\{x\in L \mid \varphi(x) \sqsubseteq x\}$.

Proof : ([5],Ch.V, dual of Th.11). Let S be the set of elements $x\in L$ such that $\varphi(x) \sqsubseteq x$. Since L is complete the greatest lower bound $\bigsqcap S$ of S exists. Let us denote $\bigsqcap S$ by a. Since $\top$ is the supremum of L, $\varphi(\top) \sqsubseteq \top$ hence S is not empty. Since φ is monotone and $a \sqsubseteq x$ for all $x\in S$, $\varphi(a) \sqsubseteq \varphi(x) \sqsubseteq x$ for all $x\in S$; hence $\varphi(a) \sqsubseteq \bigsqcap S=a$. It follows, since φ is monotone, that $\varphi(\varphi(a)) \sqsubseteq \varphi(a)$, whence $\varphi(a)\in S$.

But this implies $a \sqsubseteq \varphi(a)$ since $a = \sqcap S$. We conclude that a is a fixed point of φ. Let b be another fixed point of φ, since $\varphi(b) = b$, $b \in S$. Hence $a = \sqcap S$ implies $a \sqsubseteq b$ which proves that a is the unique least fixed point of φ. *End of Proof.*

The above theorem is due to Knaster([40]). In addition Tarski([60]) proves that the set of fixed points of φ is a complete lattice. This theorem permits the existence of a solution to the system of equations to be discussed. However it is not constructive, and additional hypothesis are necessary to provide an algorithmic definition of $\mathit{lfp}(F)$.

2.4 CONSTRUCTION OF THE LEAST SOLUTION TO THE SYSTEM OF EQUATIONS BY SUCCESSIVE APPROXIMATIONS

Example : The least solution to the system of equations :

$$\begin{cases} X_1 = \dotplus \\ X_2 = X_1 \sqcup X_3 \\ X_3 = X_2 \boxplus \dotplus \end{cases}$$

can be automatically constructed by successive approximations, as follows :

First approximation :

$$\begin{bmatrix} X_1^0 = \bot \\ X_2^0 = \bot \\ X_3^0 = \bot \end{bmatrix}$$

The second approximation is obtained by replacing X_1, X_2, X_3 by the values obtained at the first approximation in the right hand side of the system of equations. We get :

$$\begin{bmatrix} X_1^1 = \dotplus & & \\ X_2^1 = X_1^0 \sqcup X_3^0 & = & \bot \sqcup \bot = \bot \\ X_3^1 = X_2^0 \boxplus \dotplus & = & \bot \boxplus \dotplus = \bot \end{bmatrix}$$

Third approximation :

$$\begin{bmatrix} X_1^2 = \dotplus & & \\ X_2^2 = X_1^1 \sqcup X_3^1 & = & \dotplus \sqcup \bot = \dotplus \\ X_3^2 = X_2^1 \boxplus \dotplus & = & \bot \boxplus \dotplus = \bot \end{bmatrix}$$

Fourth approximation :

$$\begin{bmatrix} X_1^3 = \dotplus & & \\ X_2^3 = X_1^2 \sqcup X_3^2 & = & \dotplus \sqcup \bot = \dotplus \\ X_3^3 = X_2^2 \boxplus \dotplus & = & \dotplus \boxplus \dotplus = \dotplus \end{bmatrix}$$

At last iteration recognizes that no change occurs in the values of X_1, X_2 and X_3 :

$$\begin{bmatrix} X_1^4 = \dotplus & & \\ X_2^4 = X_1^3 \sqcup X_3^3 & = & \dotplus \sqcup \dotplus = \dotplus \\ X_3^4 = X_2^3 \boxplus \dotplus & = & \dotplus \boxplus \dotplus = \dotplus \end{bmatrix}$$

so that we have obtained the least solution to the equations.

This is certainly a toilsome but at least systematic and simply automatizable way to prove that x is positive in the program : x:=1 {1} <u>while</u> <u>true</u> <u>do</u> {2} x:=x+1 {3} <u>od</u> ;
End of Example.

DEFINITION 2.4.1 *A map* $\varphi : D \to D'$ *of a complete lattice* $(D, \sqsubseteq, \sqcup)$ *into the complete lattice* $(D', \sqsubseteq', \sqcup')$ *is called* upper-semi-continuous (*in short* continuous) *if whenever* $X = \{x_1, x_2, \ldots, x_n, \ldots\}$ *where* $X \subseteq D$ *and* $x_1 \sqsubseteq x_2 \sqsubseteq \ldots \sqsubseteq x_n \sqsubseteq \ldots$ *then* $\varphi(\sqcup X) = \sqcup' \{\varphi(x) \mid x \in X\}$.

Note that a continuous map is monotone. Conversely a monotone map φ is distributive over finite chains since $x \sqsubseteq y$ implies $\varphi(x \sqcup y) = \varphi(x) \sqcup \varphi(y)$. This is not true in general for infinite chains unless when φ is continuous.

(This notion of continuity corresponds to the topological definition, see for example [15] or [53]).

HYPOTHESIS 2.4.2 *The functions* $f_i : L^n \to L$, $i \in [1,n]$ *are assumed to be continuous.*

LEMMA 2.4.3 *The function* F *on the complete lattice* L^n *to itself is continuous.*

This is a direct outcome of the hypothesis of continuity in the f_i. The main consequence of this lemma 2.4.3. is that the least fixed point of F can be defined as *the limit of a sequence of successive approximations.*

THEOREM 2.4.4 *Every continuous function* φ *on the complete lattice* $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap, \bigsqcup, \bigsqcap)$ *to itself has a least fixed point given by the formula :* $lfp(\varphi) = \bigsqcup\{\varphi^k(a) \mid k \geq 0\}$ *where* $a \in L$ *is such that* $a \sqsubseteq \varphi(a)$ *and* $a \sqsubseteq lfp(\varphi)$ *and* φ^k *is the* k*-fold composition of* φ *with itself.*

Proof : Since L is a complete lattice $p = \bigsqcup\{\varphi^k(a) \mid k \geq 0\}$ exists.

2.4.4.1 - p is a fixed point of φ

Since $a \sqsubseteq \varphi(a)$ and φ is monotone, we can prove by recurrence on k that $\varphi^k(a) \sqsubseteq \varphi^{k+1}(a)$ holds for all k. Hence by transitivity, $a = \varphi^0(a) \sqsubseteq \varphi^k(a)$ for all $k \geq 1$. Therefore $p = \bigsqcup\{\varphi^k(a) \mid k \geq 0\} = \bigsqcup\{\varphi^k(a) \mid k \geq 1\} = \bigsqcup\{\varphi^{k+1}(a) \mid k \geq 0\} = \varphi(\bigsqcup\{\varphi^k(a) \mid k \geq 0\})$ since φ is continuous. Then $p = \varphi(p)$.

2.4.4.2 - p is the least fixed point of φ

Theorem 2.3.6. proves the existence of a least fixed point $lfp(\varphi)$ of φ. Since $a \sqsubseteq lfp(\varphi)$ and φ is monotone we can prove by induction on k that $\varphi^k(a) \sqsubseteq lfp(\varphi)$. This proves that $\bigsqcup\{\varphi^k(a) \mid k \geq 0\} \sqsubseteq lfp(\varphi)$ hence $p \sqsubseteq lfp(\varphi)$. Moreover by definition of a least fixed point $lfp(\varphi) \sqsubseteq p$, hence we conclude by antisymmetry that $p = lfp(\varphi)$. *End of Proof.*

This theorem was suggested by Tarski with the hypothesis that φ is distributive under countable joins, ([60], p.305). It is also comparable with Kleene's first recursion theorem for functional equations over integer functions, [39]. Following Scott[53], it has been proved by numerous authors with "a" trivially chosen to be the infimum "$\bot$" of L.

Notice that the least fixed point of φ is the limit $(\bigsqcup\{\varphi^k(a) \mid k \geq 0\})$ of a sequence $x^0 = a$, $x^1 = \varphi(x^0)$, ..., $x^{k+1} = \varphi(x^k)$, ... of successive approximations. This sequence forms an increasing chain, that is to say $x^0 \sqsubseteq x^1 \sqsubseteq \ldots \sqsubseteq x^k \sqsubseteq \ldots$. The iteration process eventually converges after m steps if $x^m = x^{m-1}$ (in which case $x^m = lfp(F)$). On the contrary it diverges when the sequence of successive approximations is an infinite strictly increasing chain. Therefore, one of the hypothesis which may insure *convergence* of this iterative method is that L satisfies the following ascending chain condition.

DEFINITION 2.4.5 *A partly ordered set* P *satisfies the* ascending (or descending) chain condition *if and only if all strictly ascending (descending) chains in* P *are finite.*

Example : The lattice $L = \{\dot{\pm}, \dot{+}, \dot{-}, \bot\}$ is finite, hence it satisfies the ascending and descending chain conditions ([5],Ch.VIII.1,Ex.6(a)). Since L is a lattice with infimum satisfying the ascending chain condition, it is a complete lattice ([5],Ch.VIII.1,Ex.1(a)). The equations associated with any particular program are obtained by composition of monotone elementary functions, hence the f_i are monotone. The elementary functions (constant $\dot{+}$, $\sqcup$, $\boxplus$) are shown to be monotone by case analysis.

Since F is monotone and L^3 satisfies the ascending chain condition, it is continuous. The least fixed point of F has been computed by successive approximations starting from the infimum $<\bot,\bot,\bot>$ of L^3. This iteration process necessarily converges for any program (any F) since L^n satisfies the ascending chain condition, ([5],Ch.VIII.1,Ex.4). *End of Example.*

2.5 CONSTRUCTION OF THE LEAST FIXED POINT OF F BY CHAOTIC ITERATIONS

We have solved the fixed point equation X=F(X) by Jacobi's method of successive approximations $X^{k+1}=F(X^k)$, (k=0,1,2,...) which can be detailed as :

$$\begin{cases} X_i^{k+1} = f_i(X_1^k, X_2^k, \ldots, X_n^k) & (k=0,1,2,\ldots) \\ i=1,2,\ldots,n \end{cases}$$

In practice the Gauss-Seidel's iterative method :

$$\begin{cases} X_1^{k+1} = f_1(X_1^k, X_2^k, \ldots\ldots\ldots\ldots\ldots\ldots\ldots, X_n^k) \\ \ldots \\ X_i^{k+1} = f_i(X_1^{k+1}, \ldots\ldots, X_{i-1}^{k+1}, X_i^k, \ldots\ldots, X_n^k) \\ \ldots \\ X_n^{k+1} = f_n(X_1^{k+1}, \ldots\ldots\ldots\ldots\ldots, X_{n-1}^{k+1}, X_n^k) \end{cases}$$

which consists in continually reinjecting in the computations the last results of the computations themselves would reduce the memory congestion and accelerate the convergence.

Example : Solving the system of equations

$$\begin{cases} X_1 = \dot{+} \\ X_2 = X_1 \sqcup X_3 \\ X_3 = X_2 \boxplus \dot{+} \end{cases}$$

using Gauss-Seidel's method we get :

First approximation :

$$\begin{bmatrix} X_1^0 = \bot \\ X_2^0 = \bot \\ X_3^0 = \bot \end{bmatrix}$$

Second approximation :

$$\begin{bmatrix} X_1^1 = \dot{+} \\ X_2^1 = X_1^1 \sqcup X_3^0 = \dot{+} \sqcup \bot = \dot{+} \\ X_3^1 = X_2^1 \boxplus \dot{+} = \dot{+} \boxplus \dot{+} = \dot{+} \end{bmatrix}$$

A last iteration proves stabilization :

$$\begin{bmatrix} X_1^2 = \dot{+} = X_1^1 \\ X_2^2 = X_1^2 \sqcup X_3^1 = \dot{+} \sqcup \dot{+} = \dot{+} = X_2^1 \\ X_3^2 = X_2^2 \boxplus \dot{+} = \dot{+} \boxplus \dot{+} = \dot{+} = X_3^1 \end{bmatrix}$$

The iterates converge after 3 steps instead of 5. *End of Example.*

In general Robert([50]) shows that Gauss-Seidel's methods is not algorithmicly more reliable than Jacobi's successive approximations method. This means that without sufficient hypothesis on F Jacobi's method may converge although the Gauss-Seidel one cycles. The contrary is also true, that is Gauss-Seidel's method may converge although Jacobi's iterations endless cycle. Fortunately this phenomenon is impossible when F is continuous.

We will now show that any *chaotic iteration method* converges to the least fixed point of F. Otherwise stated this signifies that one can arbitrarily determine at each step which are the components of the system of equations which will evolve and in what order (as long as no component is forgotten indefinitely).

DEFINITION 2.5.1 *Let J be a subset of* $\{1,\ldots,n\}$. *We denote by* F_J *the map* $L^n \to L^n$ *defined by :* $F_J(X_1,\ldots,X_n)=\langle Y_1,\ldots,Y_n\rangle$ *where* $\forall i\in[1,n]$ *we have :*

$$\begin{cases} Y_i = f_i(X_1,\ldots,X_n) & \text{if } i\in J \\ Y_i = X_i & \text{if } i\notin J \end{cases}$$

(As before we will go on denoting $F_{\{1,\ldots,n\}}$ by F).

DEFINITION 2.5.2 *An* ascending sequence of chaotic iterations *corresponding to the operator F and starting with a given vector* X^0 *such that* $X^0 \sqsubseteq F(X^0)$ *and* $X^0 \sqsubseteq lfp(F)$ *is a sequence* X^k, $k=0,1,\ldots$ *of vectors of* L^n *defined recursively by :* $X^k=F_{J_{k-1}}(X^{k-1})$ *where* J_k, $k=0,1,\ldots$ *is a sequence of subsets of* $\{1,\ldots,n\}$ *such that no component is forgotten indefinitely that is :* $\{\exists m>0 \mid \{(\forall i\in[1,n]),(\forall k\geq 0),(\exists \ell\in[0,m[)\mid i\in J^{k+\ell}\}\}$.

Note : The choice $J_k=\{1,\ldots,n\}$, $\forall k$ corresponds to Jacobi's iterative method, whereas the choice $J_k=\{(k \text{ \underline{modulo} } n)+1\}$, $\forall k$ corresponds to Gauss-Seidel's iterative method. *End of Note.*

THEOREM 2.5.3 *The limit* $\coprod_{k=0}^{\infty} X^k$ *of any ascending sequence of chaotic iterations* $X^0,\ldots,X^k,X^{k+1},\ldots$ *is equal to the least fixed point* $lfp(F)$ *of* F.

LEMMA 2.5.3.1 $\{\forall k\geq 0,\ X^k \sqsubseteq X^{k+1} \sqsubseteq F(X^k) \sqsubseteq lfp(F)\}$

Proof : Let us first remark that whenever $X\sqsubseteq F(X)\sqsubseteq lfp(F)$ we have $\forall J\in\{1,\ldots,n\}$, $X\sqsubseteq F_J(X)\sqsubseteq F(X) \sqsubseteq lfp(F)$. Indeed $\forall i\in[1,n]$, $X_i\sqsubseteq f_i(X)$ therefore if $i\in J$ then $X_i\sqsubseteq f_i(X)=F(X)_i=F_J(X)_i$ otherwise $X_i=F_J(X)_i\sqsubseteq f_i(X)$.

Since by hypothesis $X^0\sqsubseteq F(X^0)\sqsubseteq lfp(F)$ this implies $X^0\sqsubseteq F_{J^0}(X^0)=X^1\sqsubseteq F(X^0) \sqsubseteq lfp(F)$. For the induction step let us assume that $X^{k-1}\sqsubseteq X^k\sqsubseteq F(X^{k-1})\sqsubseteq lfp(F)$ for some k>0. If $i\in J^{k-1}$ then $X_i^k=f_i(X^{k-1})\sqsubseteq f_i(X^k)\sqsubseteq lfp(F)_i$ since $X^{k-1}\sqsubseteq X^k\sqsubseteq lfp(F)$ and f_i is monotone. Otherwise $i\notin J^{k-1}$ and $X_i^k=X_i^{k-1}\sqsubseteq f_i(X^{k-1})\sqsubseteq lfp(F)_i$ by induction hypothesis so that $X_i^k\sqsubseteq f_i(X^{k-1})\sqsubseteq f_i(X^k)\sqsubseteq lfp(F)_i$ by monotony. In both cases we have $\forall i\in[1,n]$, $X_i^k\sqsubseteq f_i(X^k)\sqsubseteq lfp(F)_i$ therefore $X^k\sqsubseteq F(X^k)\sqsubseteq lfp(F)$ proving that $X^k\sqsubseteq X^{k+1}=F_{J^k}(X^k)\sqsubseteq F(X^k)\sqsubseteq lfp(F)$. *End of Proof.*

LEMMA 2.5.3.2 $\{\exists q\in[0,m] \mid (\forall k\geq 0),\ (F(X^k)\sqsubseteq X^{k+q})\}$

Proof : The proof is by reductio ad absurdum. Let us suppose that $\{\forall q\in[0,m],\ \exists k\geq 0 \mid \underline{\text{not}}(F(X^k)\sqsubseteq X^{k+q})\}$. This is equivalent to $\{\forall q\in[0,m],\ \exists k\geq 0 \mid (X^{k+q}\sqsubset F(X^k))\ \underline{\text{or}}\ (F(X^k)$ not comparable with $X^{k+q})\}$. Suppose that $\forall q\in[0,m]$, $\exists k\geq 0$ such that $F(X^k)$ is not comparable with X^{k+q}. This must be true for q=0 which contradicts lemma 2.5.3.1. Suppose now that $\forall q\in[0,m]$, $\exists k\geq 0$ such that $X^{k+q}\sqsubset F(X^k)$, that is by definition of the strict inequality we have $X^{k+q}\sqsubseteq F(X^k)$ and $X^{k+q}\neq F(X^k)$. This implies that for some component $i\in[1,n]$ $X_i^{k+q}\sqsubset f_i(X^k)$, while for the other components the inequality is not necessarily strict. By definition of chaotic iterations $\{\exists m>0 \mid \{\forall i,\ \forall k,\ \exists \ell\in[0,m[\mid i\in J^{k+\ell}\}\}$, therefore $X_i^{k+\ell+1}=f_i(X^{k+\ell})$. But lemma 2.5.3.1 implies by transitivity that $X^k\sqsubseteq X^{k+\ell}$ thus by monotony $f_i(X^k)\sqsubseteq f_i(X^{k+\ell})$ which implies $f_i(X^k)\sqsubseteq X_i^{k+\ell+1}$. Choosing $q=\ell+1$ we have by hypothesis $\exists k$ such that $X_i^{k+q}\sqsubset f_i(X^k)$ and also $f_i(X^k)\sqsubseteq X_i^{k+q}$ which is impossible. This contradiction proves the truth of lemma 2.5.3.2. *End of Proof.*

Proof of Theorem 2.5.3 : Let us first prove that $\coprod_{k=0}^{\infty} X^k \sqsubseteq F(\coprod_{k=0}^{\infty} X^k)$. According to lemma 2.5.3.1 we have $\forall k\geq 0$, $X^k\sqsubseteq F(X^k)$. Since $\coprod$ is monotone we get $\coprod_{k=0}^{\infty} X^k \sqsubseteq \coprod_{k=0}^{\infty} F(X^k)$. The sequence

of chaotic iterations is an increasing chain (lemma 2.5.3.1) and F is continuous, hence $\bigsqcup_{k=0}^{\infty} F(X^k) = F(\bigsqcup_{k=0}^{\infty} X^k)$ hence by transitivity $\bigsqcup_{k=0}^{\infty} X^k \sqsubseteq F(\bigsqcup_{k=0}^{\infty} X^k)$. Let us now prove that $F(\bigsqcup_{k=0}^{\infty} X^k) \sqsubseteq \bigsqcup_{k=0}^{\infty} X^k$. According to lemma 2.5.3.2, $\exists q \in [0,m]$ such that $\forall k \geq 0,\ F(X^k) \sqsubseteq X^{k+q}$. Hence by monotony $\bigsqcup_{k=0}^{\infty} F(X^k) \sqsubseteq \bigsqcup_{k=0}^{\infty} X^{k+q} = \bigsqcup_{k=q}^{\infty} X^k = \bigsqcup_{k=0}^{\infty} X^k$ since $\bigsqcup_{k=0}^{q-1} X^k \sqsubseteq X^{k+q}$ (lemma 2.5.3.1). By continuity of F we deduce $F(\bigsqcup_{k=0}^{\infty} X^k) \sqsubseteq \bigsqcup_{k=0}^{\infty} X^k$, hence by antisymmetry we conclude $F(\bigsqcup_{k=0}^{\infty} X^k) = \bigsqcup_{k=0}^{\infty} X^k$.
Also lemma 2.5.3.1 implies $\bigsqcup_{k=0}^{\infty} X^k \sqsubseteq \bigsqcup_{k=0}^{\infty} \mathit{lfp}(F) = \mathit{lfp}(F)$ and by uniqueness of the least fixed point of F we conclude $\bigsqcup_{k=0}^{\infty} X^k = \mathit{lfp}(F)$. *End of Proof.*

Theorem 2.5.3 imposes to take $\mathit{lfp}(F)$ to be the join $\bigsqcup_{k=0}^{\infty} X^k$ of all terms of the chaotic iteration sequence. In practice we can overcome this difficulty thanks to the following result :

THEOREM 2.5.4 *Let m be the maximum number of steps which are necessary for any component to evolve in chaotic iterations. There exists an ordinal k of cardinality less or equal to that of* L^n *such that* $\{\forall \ell \geq km,\ \mathit{lfp}(F) = X^\ell\}$.

LEMMA 2.5.4.1 *Let* $k, \ell \geq 0$ *such that* $k \neq \ell$ *then* $\{(\mathit{lfp}(F) \neq X^{km})$ <u>or</u> $(\mathit{lfp}(F) \neq X^{\ell m})\} \Rightarrow \{X^{km} \neq X^{\ell m}\}$.

Proof : We prove that $X^{km} = X^{\ell m}$ implies $\mathit{lfp}(F) = X^{km}$ for $k < \ell$ (since the case $k > \ell$ is symmetric). According to lemma 2.5.3.1 and 2.5.3.2 we have $X^{km} \sqsubseteq F(X^{km}) \sqsubseteq X^{km+m} \sqsubseteq X^{\ell m} = X^{km}$ proving that $X^{km} = F(X^{km})$ and since $X^{km} \sqsubseteq \mathit{lfp}(F)$ we have $X^{km} = \mathit{lfp}(F)$. *End of Proof.*

Proof of Theorem 2.5.4. The proof that $\{\exists k \mid (k \leq \bar{\bar{L}}^n)$ <u>and</u> $(\ell \geq km \Rightarrow \mathit{lfp}(F) = X^\ell)\}$ is by reductio ad absurdum. Indeed, suppose that $\{\forall k,\ (k > \bar{\bar{L}}^n)$ <u>or</u> $(\ell \geq km$ <u>and</u> $\mathit{lfp}(F) \neq X^\ell)\}$. Let α be the least ordinal of cardinality strictly greater than $\bar{\bar{L}}^n$. $\forall k < \alpha$ we must have $(\ell \geq km$ <u>and</u> $\mathit{lfp}(F) \neq X^\ell)$, that is $\mathit{lfp}(F) \neq X^{km}$ when choosing ℓ to be km. Let us define $\psi \in \alpha \to L^n$ by $\psi(k) = X^{km}$. $\forall k_1, k_2 \in \alpha$ such that $k_1 \neq k_2$ (with eventually $(k_1 = \alpha)$ exclusive or $(k_2 = \alpha)$) we have $(\mathit{lfp}(F) \neq X^{k_1 m})$ or $(\mathit{lfp}(F) \neq X^{k_2 m})$ hence lemma 2.5.4.1 implies that $X^{k_1 m} \neq X^{k_2 m}$ proving that ψ is a one to one correspondence of α into L^n. Therefore α is of cardinality less or equal to that of L^n which is the desired contradiction. *End of Proof.*

Notes : (i) These theorems can be extended to asynchronous iterations as defined in [4]. (ii) When the iterating order which is used to solve the equations corresponds to the program control graph the successive approximations can be intuitively comprehended as a symbolic execution of the program. This was the way iterative methods were first understood (e.g. [12], [36], [51], [56], [65]). (iii) The question of optimal order of iteration has not yet received a conceptual answer. (e.g. [2], [33], [59]). *End of Notes.*

2.6 TYPOLOGY OF ABSTRACT INTERPRETATIONS

2.6.1 DUAL ABSTRACT INTERPRETATIONS

By the *Duality Principle* ([5], p.3), we can replace all occurrences of ($\sqsubseteq$, $\bot$, $\top$, $\sqcup$, $\sqcap$, $\coprod$, $\prod$, *ascending*, *least fixed point*(lfp)) respectively by ($\sqsupseteq$, $\top$, $\bot$, $\sqcap$, $\sqcup$, $\prod$, $\coprod$, *descending*, *greatest fixed point*(gfp)) to get dual results. These dual results are never stated explicitly but are always implicit for all statements in this paper.

2.6.2 FORWARD AND BACKWARD SYSTEMS OF EQUATIONS

We defined the invariant P_i at point i as a function $f_i(P_1, \ldots, P_n)$ of the invariants associated with each of the program points $\{1\}, \ldots, \{n\}$. Two particular cases are of special practi-

cal importance. In *forward equations* each invariant P_i is only function of the invariants associated with the program points {j} which precede {i} in the control flow (e.g. [19]), whereas in *backward equations* each invariant P_i is only a function of the invariants associated with the program points {k} which follow {i} in the control flow (e.g. live expressions at paragraph 7.1). The *mixed case* where each invariant P_i is a function of the invariants associated with the program points which either precede or follow {i} in the control flow is also widely used (e.g. [61]).

2.7 DISCUSSION ON ALTERNATE MATHEMATICAL MODELS

Arbitrary posets are not in general complete lattices. Other well-known fixed point theorems might be used in such a case (cf. [1], [28] etc.). Other convenient algebras permit constructive definitions of fixed points to be given (cf. chain complete partly ordered sets, complete ordered F-magna [47], [11], initial continuous algebras [20]). However we choose to use the complete lattice model because it is well-known. Moreover any poset can be made a complete lattice by known systematic methods (cf. [42]).

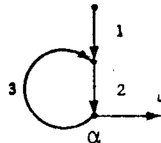
3. APPLICATION TO PERFORMANCE ANALYSIS OF PROGRAMS

This example of application is presented first, since it allows the mathematical model which is used to be understood by analogy with numerical analysis techniques.

3.1 ASSOCIATING A SYSTEM OF EQUATIONS WITH A PROGRAM

The performance of a program may be analyzed by counting the number of times each step of the program is executed.

We will model the program by a connected directed graph with a single entry arc and whose nodes are junction, branch or separation (test) points.



Suppose we are given for each test α in the program the probability $p(\alpha)$ that this test will be true after being evaluated. It may be very difficult to obtain an exact expression of these probabilities in terms of known properties of the input, for example internal tests may depend on computed quantities having no simple relation to the input. A major simplification is to consider tests as Markov processes, i.e. the probability is constant and independent of prior history. Furthermore we assume these probabilities to be "given" (e.g. as determined by measurements).

We wish to determine the expected frequency X_i of traversing each arc i in the program during a single execution of the whole program. The expected frequencies are given by the solution to a system of equations generated from the program by application of Kirchhoff's first law of conservation of flow :

$$\begin{cases} X_1 = 1 \\ X_2 = X_1 + X_3 \\ X_3 = X_2 . p(\alpha) \\ X_4 = X_2 . (1-p(\alpha)) \end{cases}$$

3.2 THE COMPLETE LATTICE OF FREQUENCIES

Frequencies are zero or positive reals $\mathbb{R}^+$ ordered by the natural ordering $\leq$ ($\sqsubseteq$). The least upper bound operation $\sqcup$ is the maximum $\underline{max}$, and the greatest lower bound operation $\sqcap$ is the mimimum $\underline{min}$ operation. The infimum $\perp$ of the poset $\mathbb{R}^+$ is equal to $0=\underline{MIN}\{i \mid i\in\mathbb{R}^+\}$. Notice that $\mathbb{R}^+$ is conditionally complete ([5], Ch.V, §3) but not complete since the expression $\underline{MAX}\{i \mid i\in\mathbb{R}^+\}$ is not defined. However we can make $\mathbb{R}^+$ into a complete lattice $\mathbb{R}^*$ by adjoining a supremum $\top$ denoted ∞ and defined by $\infty=\underline{MAX}\{i \mid i\in\mathbb{R}^+\}$. Now, $\mathbb{R}^*(\leq,0,\infty,\underline{max},\underline{min},\underline{MAX},\underline{MIN})$ is a complete lattice.

3.3 SOLVING THE SYSTEM OF EQUATIONS

Let us simplify the system of equations by elimination of the variables X_1 and X_3 (and supposing that the value of $p(\alpha)$ is given by a constant expression p).

$$\begin{cases} X_2 = 1+X_2.p \\ X_4 = X_2.(1-p) \end{cases}$$

Since X_2 depends only on itself we can first solve the subsystem $X_2=1+X_2.p$, (i.e. $X_2=F(X_2)$ where $F(X)=1+X.p$).

It is obvious that the solutions to this equation in $\mathbb{R}^*$ are $1/(1-p)$ and ∞. However going on with this example provides an intuitive application of theorems 2.3.6 and 2.5.4 .

3.3.1 EXISTENCE OF SOLUTIONS

Theorem 2.3.6 requires F to be order-preserving which is obvious since $p\geq 0$.

Note : The proof that F is monotone need not be done for every particular program. In general it is possible to show that the isotony (as well as continuity) of F is a direct consequence of the syntactic method which is utilized to built the system of equations. *End of Note.*

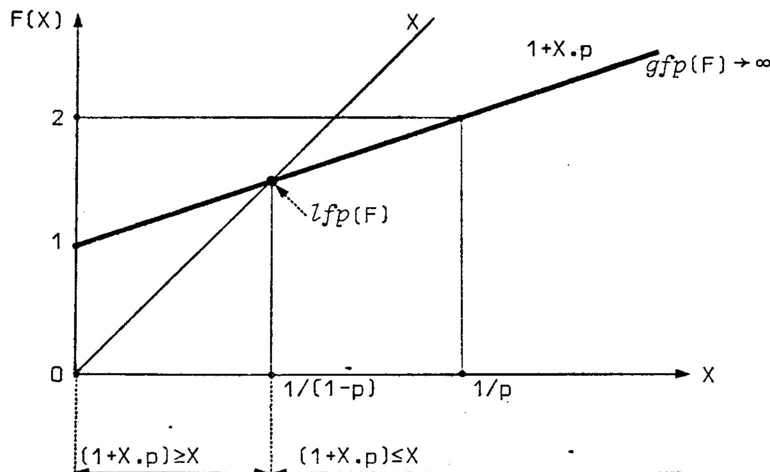
Theorem 2.3.6 then states that the extreme fixed points of F are :

$$\begin{aligned} lfp(F) &= \underline{MIN}\{X\in\mathbb{R}^* \mid (1+X.p)\leq X\} \\ &= \underline{MIN}\{X \mid (1/(1-p))\leq X\leq\infty\} \\ &= 1/(1-p) \end{aligned}$$

Dually :

$$\begin{aligned} gfp(F) &= \underline{MAX}\{X\in\mathbb{R}^* \mid X\leq(1+X.p)\} \\ &= \underline{MAX}(\{X \mid 0\leq X\leq(1/(1-p))\}\cup\{\infty\}) \\ &= \infty \end{aligned}$$

These results are easily understood by the following geometric interpretation :



3.3.2 CONSTRUCTION OF THE EXTREME SOLUTIONS

Note that the previous definition of the fixed points was not constructive, whereas theorem 2.4.5 provides an algorithmic construction by successive approximations.

The map F is clearly continuous since it is infinitely distributive that is for any arbitrary indexing set Δ we have :

$1+\underline{MAX}\{X_i \mid i\in\Delta\}.p = \underline{MAX}\{1+X_i.p \mid i\in\Delta\}$

- The descending approximation sequence leads to the maximal fixed point :

$X^0 = \infty$

$X^1 = F(X^0) = 1+\infty.p = \infty$

$= X^0$

- The ascending approximation sequence leads to the minimal fixed point :

$X^0 = 0$

$X^1 = F(X^0) = 1+0.p = 1$

$X^2 = F(X^1) = 1+1.p = 1+p$

$X^3 = F(X^2) = 1+(1+p)+p = 1+p+p^2$

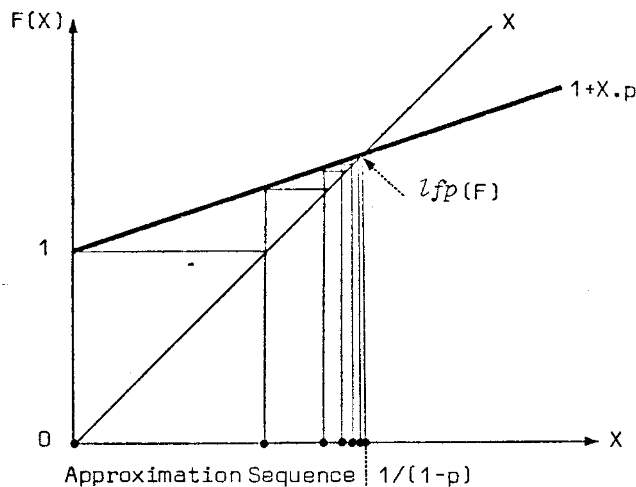
...

$X^k = F(X^{k-1}) = 1+p+p^2+\ldots+p^{k-1}$

...

The limit of the ascending approximation sequence is an infinite series, the sum of which is $lfp(F)=1/(1-p)$.

- The classical geometric interpretation is the following :



4. COPING WITH INFINITE APPROXIMATION SEQUENCES

When an approximation sequence is infinite, it is impossible to use an iterative resolution method to compute its limit that is the exact solution to the system of equations. In fact the problem of mechanically computing the least solution $S\in L^n$ of the equations $X=F(X)$ is in general undecidable, [29]. This does not rule out finding algorithms for computing $S=lfp(F)$ for particular F and L. Further, it is fundamental to note that an *approximation* of the exact least solutions to mechanically unsolvable systems of equations can always be automatically computed [13].

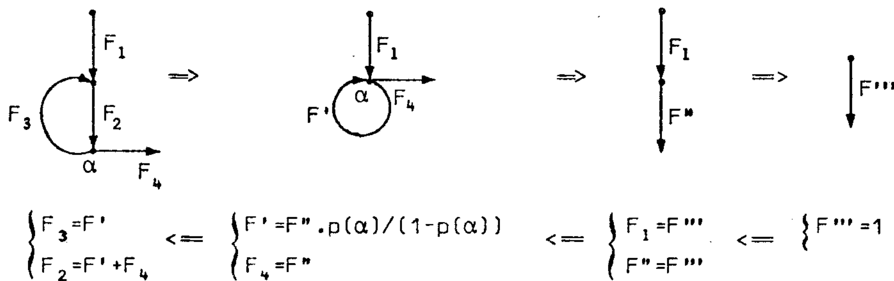
We can roughly classify the methods for coping with infinite approximation sequences as follows :

4.1 DIRECT RESOLUTION METHODS

4.1.1 FORMAL RESOLUTION

When F and L possess the necessary algebraic properties the system of equations X=F(X) may be solved formally by eliminations and simplifications.

Example : The linear equations obtained in the "performance analysis of programs" may be solved by formal substitutions, successively applying simplification rules until obtaining equations whose solution is known. This is the method commonly used when solving equations by hand. This formal resolution process is in general presented as a sequence of reductions of the program graph by elementary transformations (e.g. [35]) :



The method is applicable only when considering appropriate applications (using algebras allowing the above formal manipulations) and appropriate programs which permit a simple simplification algorithm (i.e. the program flow graph must be "reducible"). *End of Example.*

4.1.2 SYSTEM OF DIFFERENCE EQUATIONS

Let $S^0, S^1, \ldots S^k, \ldots$ be the sequence of successive approximations converging to the solution S to the system of equations X=F(X). This sequence is defined recursively by $S^{k+1}=F(S^k)$. Knowing S^0, the system of difference equations $S^{k+1}=F(S^k)$ may possibly be solved to get S^k as a function S of k, $S^k=S(k)$. The solution S to the equations is then $\lim_{k\to\infty} S(k)$.

Example : The linear equations obtained in the "performance analysis of programs" may be expressed as difference equations ([64]) which may be automatically solved, [10]. For example the equation X=1+X.p has a solution X(∞) defined by :

$$\begin{cases} X(0) = 0 \\ X(k+1) = 1+X(k).p \end{cases}$$

These simple difference equations have the solution : $X(k)=(1+p^k)/(1-p)$ and $\lim_{k\to\infty} p^k=0$ when $0\le p<1$ in which case $X(\infty)=1/(1-p)$, otherwise p=1 and then $X(\infty)=\infty$. *End of Example.*

4.2 VERIFICATION OF PROPERTIES OF THE SOLUTION

Generally some properties of programs may be proved to hold without full knowledge of the solution S to the system of equations X=F(X). It suffices to prove that some property P(S) holds for S. Since the solution S to the system of equations is defined as the limit of an approximation sequence S^0, $S^1=F(S^0), \ldots, S^{k+1}=F(X^k), \ldots$ one can prove P(S) using Scott's induction rule :

From $\{P(S^0)$ <u>and</u> $\{\{(\forall X)\ P(X)\} \Rightarrow \{P(F(X))\}\}\}$ infer P(S).

(Recall that S^0 must be chosen such that $S^0 \sqsubseteq F(S^0)$ and $S^0 \sqsubseteq lfp(F)$ so that $S^0 = \bot^n$ is a convenient choice. Also P must be an "admissible" predicate (see e.g. [43])) remaining true when passing to the limit. Rigorously we should apply the second principle of transfinite induction, [5]. Other induction principles ([6],[17],[45],etc.) can be derived from Scott's induction rule, [63].

4.3 CONSTRUCTION OF APPROXIMATE SOLUTIONS

Generally, the above methods are not fully automatizable since it is undecidable to solve the system of equations. However, it is always possible to mechanically compute an approximation of the exact solution. The approximate solution will provide useful if not perfect information.

Example : The application of the rules of signs to programs provides an approximate analysis of these programs. For example, the system of equations corresponding to the program

x:=1; {1} <u>while</u> <u>true</u> <u>do</u> {2} x:=x-2 {3} <u>od</u>;

is :

$$\begin{cases} X_1 = \dot{+} \\ X_2 = X_1 \sqcup X_3 \\ X_3 = X_2 \ominus \dot{+} \end{cases} \quad \text{the least solution of which is :} \quad \begin{bmatrix} X_1 = \dot{+} \\ X_2 = \dot{\pm} \\ X_3 = \dot{\pm} \end{bmatrix}$$

The fact that x is negative at program point {3} is not captured because of the rule : $\dot{+} \ominus \dot{+} = \dot{\pm}$
A more careful analysis taking account of the absolute value of x would be necessary to discover this fact. For example, we might have used the following equations over predicates :

$$\begin{cases} P_1 = (x=1) \\ P_2 = P_1 \text{ or } P_3 \\ P_3 = \{\exists x' \mid P_2(x') \text{ and } (x=x'-2)\} \end{cases} \quad \text{the solution of which is :} \quad \begin{bmatrix} P_1 = (x=1) \\ P_2 = (x \in \{-2k+1 \mid k \geq 0\}) \\ P_3 = (x \in \{-2k-1 \mid k \geq 0\}) \end{bmatrix}$$

However the approximation sequence the limit of which is the above solution is infinite. Therefore we can consider that the application of the rule of signs is a way of approximating the exact domain of x. The approximation is correct, since finding that the sign of x at point {3} is $\dot{\pm}$ corresponds to the predicate $(-\infty \leq x \leq +\infty)$ which is implied by $(x \in \{-2k-1 \mid k \geq 0\})$, so that none of the possible states of x at point {3} during execution have been left out. On the contrary, some impossible states such as x=5 have been predicted by the rule of signs, but this is precisely where the approximation took place.

Notice that the same idea of approximation is essential in other abstract interpretations such as casting out of nines in arithmetic, parity checks in hardware, dimensional analysis in physics... : these techniques permit automatic verification of sufficient (but in general not necessary) conditions of the truth or falsehood of a property,[56]. *End of Example.*

DEFINITION 4.3.0.1 *A property* $\overline{P} \in (L, \sqsubseteq)$ *is said to* correctly approximate *a property* $P \in L$ *if and only if* $P \sqsubseteq \overline{P}$.

THEOREM 4.3.0.2 *Any post-fixed point* X of F, *that is such that* $F(X) \sqsubseteq X$ *correctly approximates* $lfp(F)$.

Proof : According to theorem 2.3.6 $lfp(F) = \sqcap\{Y \in L^n \mid F(Y) \sqsubseteq Y\}$. But since $F(X) \sqsubseteq X$, X belongs to the set $\{Y \in L^n \mid F(Y) \sqsubseteq Y\}$ so that it is greater than the greatest lower bound of this set, hence $lfp(F) \sqsubseteq X$. *End of Proof.*

It is always possible to mechanically compute a post-fixed point of F. The proof is that it suffices to take the supremum of L^n but this choice is of no practical interest.

We now introduce fixed-point approximation methods which lead to more accurate results.

Since the space L of *concrete properties* generally contains infinite and complex objects, it may be wise to choose a more simple space $\overline{L}$ of *abstract properties* which is a simplified image of L. Then the system of equations $X=F(X)$ on L^n can be modelled in $\overline{L}^n$ by an approximate system of equations $\overline{X}=\overline{F}(\overline{X})$. The idea is that $\overline{L}$ and $\overline{F}$ can be chosen simple enough to allow an easy computation of a correct approximation of $lfp(F)$.

Let us first establish the correspondence between the abstract properties $\overline{L}$ and the concrete properties L by a *concretization function* Γ which gives the concrete form of any abstract predicate.

HYPOTHESIS 4.3.0.3 (a) - $L(\subseteq,\bot,\top,\sqcup,\sqcap,\coprod,\prod)$ *is a complete lattice,*
(b) - $\overline{L}(\overline{\sqsubseteq})$ *is a partly ordered set,*
(c) - $\Gamma:\overline{L}\to L$ *is monotone.*

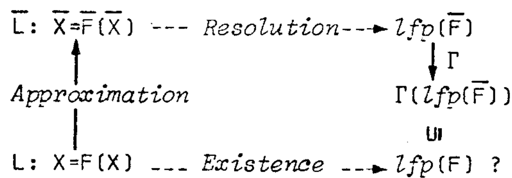
The correspondence between F and $\overline{F}$ is established by :

HYPOTHESIS 4.3.0.4 (a) - $F:L^n\to L^n$ *is monotone,* $\overline{F}:\overline{L}^n\to\overline{L}^n$
(b) - $\forall\overline{X}\in\overline{L}^n,\ F(\Gamma(\overline{X}))\subseteq\Gamma(\overline{F}(\overline{X}))$.

As usual we "componentwise" extend Γ to $\overline{L}^n$ by the definition $\Gamma(<X_1,\ldots,X_n>) = <\Gamma(X_1),\ldots,\Gamma(X_n)>$. Therefore the distinction between $\Gamma:\overline{L}\to L$ and $\Gamma:\overline{L}^n\to L^n$ is made by context.

4.3.1 STRUCTURAL APPROXIMATION

When the system of equations $\overline{X}=\overline{F}(\overline{X})$ can be solved, the following schema is of practical interest :



The solution $lfp(F)$ of the equations $X=F(X)$ exists but cannot be mechanically computed. However an approximate system of equations $\overline{X}=\overline{F}(\overline{X})$ may be associated with any F and solved. The following theorem ensures that $lfp(F)\subseteq\Gamma(lfp(\overline{F}))$ so that the concrete form $\Gamma(lfp(\overline{F}))$ of the abstract solution $lfp(\overline{F})$ correctly approximates the inaccessible concrete solution $lfp(F)$.

THEOREM 4.3.1.1 *If* $\overline{L}$ *is a complete lattice and* $\overline{F}$ *is monotone then* $\{lfp(F)\subseteq\Gamma(lfp(\overline{F}))\}$.

Proof : The existence of $lfp(F)$ and $lfp(\overline{F})$ is stated by theorem 2.3.6. Applying hypothesis 4.3.0.4.b for $\overline{X}=lfp(\overline{F})$ we get $F(\Gamma(lfp(\overline{F})))\subseteq\Gamma(\overline{F}(lfp(\overline{F}))) = \Gamma(lfp(\overline{F}))$. Since $\Gamma(lfp(\overline{F}))$ is a post-fixed-point of F, theorem 4.3.0.2 implies that $lfp(F)\subseteq\Gamma(lfp(\overline{F}))$. *End of Proof.*

Example : The automatic analysis of the program :

{1} x:=0; {2} <u>while</u> x≤n <u>do</u> {3} x:=x+2 {4} <u>od</u>; {5}

can be done by characterizing the set of states <x,n> at each program point. Therefore we have to solve the equations :

$$\begin{cases} P_1 = \{<\Omega,\alpha>\} & \text{(x is uninitialized, the initial state of n is } \alpha) \\ P_2 = \{<0,n> \mid <x,n>\in P_1\} \\ P_3 = \{<x,n>\in(P_2\cup P_4) \mid (x\leq n)\} \\ P_4 = \{<x+2,n> \mid <x,n>\in P_3\} \\ P_5 = \{<x,n>\in(P_2\cup P_4) \mid (x>n)\} \end{cases}$$

Properties are sets of couples $\langle x,n\rangle$ where x and n are integers or equal to Ω symbolizing the uninitialized value. Hence the set L of properties is the powerset of $\{\langle x,n\rangle \mid x,n\in(\mathbb{I}\cup\{\Omega\})\}$, it is a complete lattice for ordering $\subseteq$ (set inclusion) and join $\cup$ (set union).

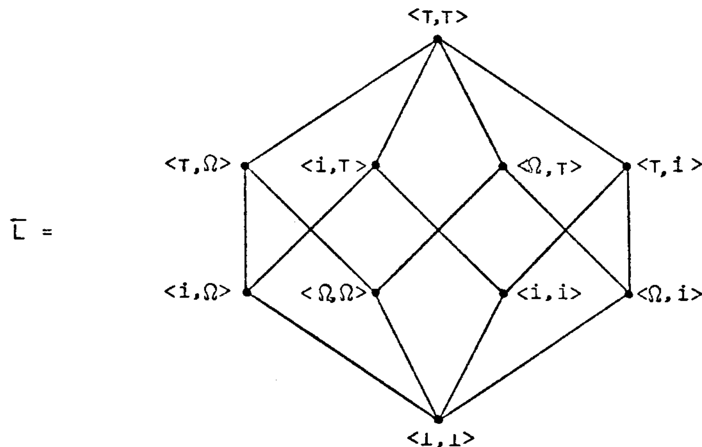
The right hand side of equations are functions of the form :

$f(P_1,P_2) = \{\varphi(X) \mid (X\in(P_1\cup P_2)) \text{ \underline{and} } \psi(X)\}$.

Supposing $((P_1\subseteq P_1') \text{ \underline{and} } (P_2\subseteq P_2'))$ we have $((X\in(P_1\cup P_2)) \text{ \underline{and} } \psi(X))$ implies $((X\in(P_1'\cup P_2')) \text{ \underline{and} } \psi(X))$, so that $\{X \mid (X\in(P_1\cup P_2)) \text{ \underline{and} } \psi(X)\} \subseteq \{X \mid (X\in(P_1'\cup P_2')) \text{ \underline{and} } \psi(X)\}$. Since $\varphi(X)$ is always defined (by taking the convention that $\Omega+2=\Omega$ and no overflow can occur) this implies $f(P_1,P_2)\subseteq f(P_1',P_2')$ so that the function f is monotone.

The system of equations has a least solution, which can be computed by successive approximations starting from the initial approximation $P_1=P_2=P_3=P_4=P_5=\emptyset$. In fact the development of this approximation sequence is almost identical to program execution, which in practice cannot be considered as a static analysis of the program.

- If we are interested in initialization problems, we can represent the set of states by the abstract lattice :



the meaning of which is given by the concretization function :

$\Gamma(\langle\bot,\bot\rangle) = \emptyset$

$\Gamma(\langle\bar{x},\bar{n}\rangle) = \{\langle x,n\rangle \mid x\in\gamma(\bar{x}) \text{ \underline{and} } n\in\gamma(\bar{n})\}$

where $\gamma(\Omega)=\{\Omega\}$, $\gamma(i)=\mathbb{I}$, $\gamma(\top)=\mathbb{I}\cup\{\Omega\}$ and $\mathbb{I}$ is the set of integers.

Taking the point of view that tests have no influence on initialization (that is it is not defined whether a test involving an uninitialized variable will be true or false at execution) the system of abstract equations is :

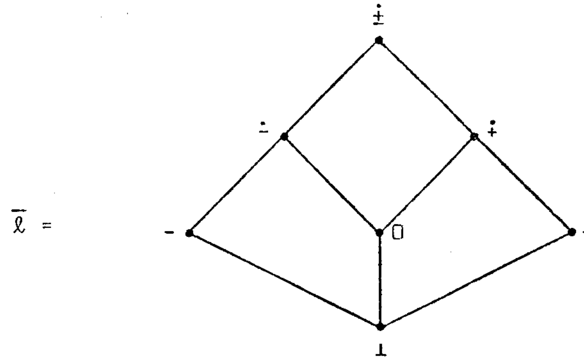
$$\begin{cases} P_1 = \langle\Omega,i\rangle & \text{(n is initialized on program entry)} \\ P_2 = \langle i,P_1.n\rangle \\ P_3 = P_2 \bar{\sqcup} P_4 \\ P_4 = \langle P_3.x \oplus i,\ P_3.n\rangle \\ P_5 = P_2 \bar{\sqcup} P_4 \end{cases}$$

This system can be solved by successive approximations starting from initial approximation $P_1^0=P_2^0=P_3^0=P_4^0=P_5^0=\langle\bot,\bot\rangle$. The solution is : $P_1=\langle\Omega,i\rangle$, $P_2=P_3=P_4=P_5=\langle i,i\rangle$.

Notice that the problem of determining which variables of a program are initialized is undecidable, so that the answer which is given to this problem is either "yes", "no", "this program point is unreachable during execution", or finally "we don't know !". These answers correspond respectively to the values i, Ω, $\bot$ and $\top$.

– If we are interested in determining the sign of integer variables we can use the lattice $\overline{L}=\overline{\ell}^2$, where $\overline{\ell}$ is defined by the following Hasse diagram :

$\overline{\ell}$ =



The meaning is the following :

$\Gamma(<\overline{x},\overline{n}>) = \{<x,n> \mid x\in\gamma(\overline{x}) \text{ and } n\in\gamma(\overline{n})\}$

where $\gamma(\bot)=\{\Omega\}$, $\gamma(-)=\{\Omega,-1,-2,\ldots\}$, $\gamma(0)=\{\Omega,0\}$, $\gamma(+)=\{\Omega,1,2,\ldots\}$, $\gamma(\dot{-})=\{\Omega,0,-1,-2,\ldots\}$, $\gamma(\dot{+})=\{\Omega,0,1,2,\ldots\}$, $\gamma(\dot{\pm})=\mathbb{Z}\cup\{\Omega\}$. Note that the results on the signs of variables will always be conditional to a correct initialization.

The system of abstract equations is :

$$\begin{cases} P_1 = <\bot,\alpha> \\ P_2 = <0,P_1.n> \\ P_3' = P_2 \,\overline{\sqcup}\, P_4 \\ P_3 = \underline{\text{x-less-than-or-equal-to-n}}(P_3') \\ P_4 = <P_3.x \oplus +,\ P_3.n> \\ P_5 = \underline{\text{n-less-than-x}}(P_3') \end{cases}$$

It takes account of the tests, and the functions x-less-than-or-equal-to-n and n-less-than-x can be determined by case analysis, using hypothesis 4.3.0.4.b :

$\{<x,n> \mid <x,n>\in\Gamma(<\overline{x},\overline{n}>) \text{ and } (n<x)\} \subseteq \Gamma(\underline{\text{n-less-than-x}}(<\overline{x},\overline{n}>))$.

Hence if $\underline{\text{n-less-than-x}}(<\overline{x},\overline{n}>)=<\overline{x}',\overline{n}'>$, then $\overline{x}'$ and $\overline{n}'$ can be chosen to be the least elements of $\overline{\ell}$ such that :

$\forall x,n : \{x\in\gamma(\overline{x}) \text{ and } n\in\gamma(\overline{n}) \text{ and } (n<x)\} \Rightarrow \{x\in\gamma(\overline{x}') \text{ and } n\in\gamma(\overline{n}')\}$

For example, we get : $\underline{\text{n-less-than-x}}(<\dot{\pm},->) = <\dot{\pm},->$

$\underline{\text{n-less-than-x}}(<\dot{\pm},\dot{+}>) = <+,\dot{+}>$

$\underline{\text{n-less-than-x}}(<\dot{-},\dot{\pm}>) = <\dot{-},->$

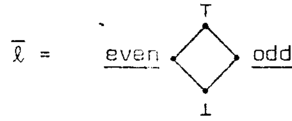
$\underline{\text{n-less-than-x}}(<-,\dot{+}>) = <\bot,\bot>$

etc.

The system of equations can be solved by successive approximations starting from the initial approximation $P_1^0=P_2^0=P_3^0=P_3'^0=P_4^0=P_5^0=\bot$. The result depends on the initial value $\alpha\in\overline{\ell}$. We get :

α	$\bot,\dot{+}$	0	$-$	$+$	$\dot{-}$	$\dot{\pm}$
P_1	$<\bot,\alpha>$	$<\bot,0>$	$<\bot,->$	$<\bot,+>$	$<\bot,\dot{-}>$	$<\bot,\dot{\pm}>$
P_2	$<0,\alpha>$	$<0,0>$	$<0,->$	$<0,+>$	$<0,\dot{-}>$	$<0,\dot{\pm}>$
P_3	$<\dot{+},\alpha>$	$<0,0>$	$<\bot,\bot>$	$<\dot{+},+>$	$<0,0>$	$<\dot{+},\dot{+}>$
P_4	$<\dot{+},\alpha>$	$<+,0>$	$<\bot,\bot>$	$<\dot{+},+>$	$<+,0>$	$<\dot{+},\dot{+}>$
P_5	$<\dot{+},\alpha>$	$<+,0>$	$<0,->$	$<+,+>$	$<\dot{+},\dot{-}>$	$<\dot{+},\dot{\pm}>$

– One can also imagine parity determination which uses the abstract lattice $\bar{L} = \bar{\ell}^2$, where :



The meaning of these abstract properties is determined by :

$\Gamma(\langle\bar{x},\bar{n}\rangle) = \{\langle x,n\rangle \mid x\in\gamma(\bar{x})$ and $n\in\gamma(\bar{n})\}$

where $\gamma(\bot)=\{\Omega\}$, $\gamma(\underline{even})=\{\Omega\}\cup\{\ldots-2,0,2,\ldots\}$, $\gamma(\underline{odd})=\{\Omega\}\cup\{\ldots-3,-1,1,3,\ldots\}$, $\gamma(\top)=\{\Omega\}\cup\mathbb{I}$.

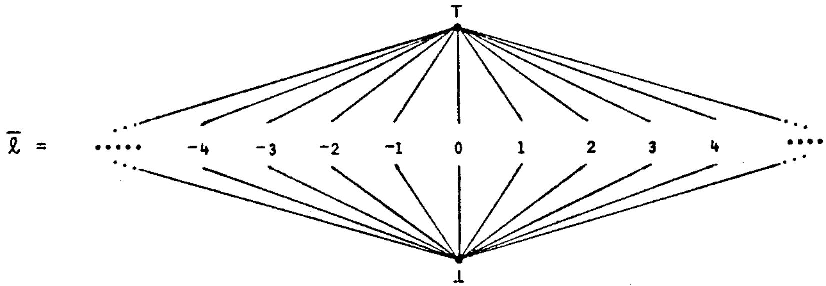
The corresponding abstract system of equations is :

$$\begin{cases} P_1 = \langle\bot,\alpha\rangle \\ P_2 = \langle\underline{even},P_1.n\rangle \\ P_3 = P_2 \bar{\sqcup} P_4 \\ P_4 = \langle P_3.x \boxplus \underline{even},\ P_3.n\rangle \\ P_5 = P_2 \bar{\sqcup} P_4 \end{cases}$$

Note that the test $(x\leq n)$ brings no information on the parity of x and n. Because of the parity rule "$\underline{even} \boxplus \underline{even} = \underline{even}$", the least solution is :

$P_1=\langle\bot,\alpha\rangle$, $P_2=P_3=P_4=P_5=\langle\underline{even},\alpha\rangle$

– A fairly classical case of program analysis and optimization occurs when constant computations are evaluated at compile time. For a program using m variables the lattice of abstract properties is $\bar{L} = \bar{\ell}^m$ where :



The meaning is given by :

$\Gamma(\langle\bar{x},\bar{n}\rangle) = \{\langle x,n\rangle \mid x\in\gamma(\bar{x})$ and $n\in\gamma(\bar{n})\}$

where $\gamma(\bot)=\{\Omega\}$, $\gamma(i)=\{i,\Omega\}$ for any integer i and $\gamma(\top)=\mathbb{I}\cup\{\Omega\}$. The system of equations is :

$$\begin{cases} P_1 = \langle\bot,\alpha\rangle \\ P_2 = \langle 1,P_1.n\rangle \\ P_3 = P_2 \bar{\sqcup} P_4 \\ P_4 = \langle P_3.x \boxplus 2,\ P_3.n\rangle \\ P_5 = P_2 \bar{\sqcup} P_4 \end{cases}$$

The union $\bar{\sqcup}$ of the lattice $\bar{\ell}$ is defined by $(\top\bar{\sqcup}x=x\bar{\sqcup}\top=\top,\ \forall x\in\bar{\ell})$, $(\bot\bar{\sqcup}x=x\bar{\sqcup}\bot=x,\ \forall x\in\bar{\ell})$, $(i\bar{\sqcup}j =$ $\underline{if}$ $i=j$ $\underline{then}$ i $\underline{else}$ $\top$ $\underline{fi}$, $\forall i,j\in\mathbb{I})$. The abstract addition operator $\boxplus$ is defined by : $(\bot\boxplus x=x\boxplus\bot=\bot,\ \forall x\in\bar{\ell})$, $(\top\boxplus x=x\boxplus\top=\top,\ \forall x\in(\bar{\ell}-\{\bot\}))$, $(i\boxplus j=i+j,\ \forall i,j\in\mathbb{I})$. Notice that $\bar{\ell}$ is infinite but satisfies the ascending chain condition (definition 2.4.5.) so that any ascending sequence of chaotic iterations is guaranteed to converge. The solution to the above equations $P_1=\langle\bot,\alpha\rangle$, $P_2=\langle 1,\alpha\rangle$, $P_3=P_4=P_5=\langle\top,\alpha\rangle$ proves that n is constant equal to its initial value α.

It is interesting to note that the determination of the constant computations in a program is undecidable so that the above interpretation is necessarily approximate. For example, the

application of this interpretation to the following skeletal program :

```
(v:=1, w:=2, x:=3, y:=3, z:=0);
while ... do
   (w:=2*v, y:=y+1, z:=z-v);
   (v:=w-v, x:=y+z);
od;
```

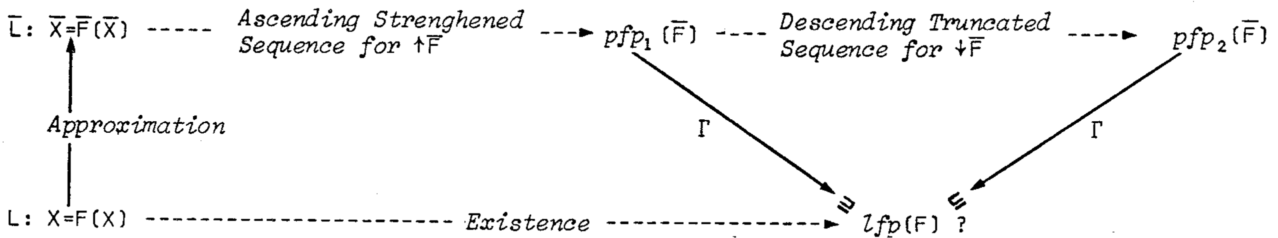
would determine that v and w are constants equal to 1 and 2 whereas x would not be found to be constant. *End of Example.*

Note : In practice, one particular interpretation of the program is considered to be its semantics. Then structural approximation provides a framework to prove that (more abstract) interpretations are correct with respect to this semantics. Also, hypothesis 4.3.0.4 is not verified for each particular program, but instead is shown to be a direct consequence of the syntactic process used to construct the system of equations for each particular program. Then, a proof that hypothesis 4.3.0.3 and 4.3.0.4 hold for any program ensures correctness. *End of Note.*

4.3.2 COMPUTATIONAL APPROXIMATION

The idea of *structural approximation* is generalized by *computational approximation*. The space of properties L is modeled by a simplified abstract space $\bar{L}$, so that the equations $X=F(X)$ in L can be reformulated by $\bar{X}=\bar{F}(\bar{X})$ in $\bar{L}$. Computational approximation then consists in computing a post-fixed-point $\bar{S}$ of $\bar{F}$. Thus hypothesis 4.3.0.3.c and 4.3.0.4.b imply $F(\Gamma(\bar{S})) \sqsubseteq \Gamma(\bar{S})$ so that according to theorem 4.3.0.2 $lfp(F) \sqsubseteq \Gamma(\bar{S})$ and therefore $\Gamma(\bar{S})$ correctly approximates the exact but unreachable $lfp(F)$. Note that structural approximation is a particular case where $\bar{S}=lfp(\bar{F})$. The point is that the computation of a post-fixed-point of $\bar{F}$ does not require $\bar{L}$ to be a complete lattice neither $\bar{F}$ to be continuous not even monotone.

We now introduce a post-fixed-point computation method. A post-fixed-point $pfp_1(\bar{F})$ of $\bar{F}$ can be computed by successive approximations as the limit of an *ascending strenghened sequence for a strenghening operator* $\uparrow\bar{F}$. Then we show how a post-fixed-point $pfp_1(\bar{F})$ of $\bar{F}$ can possibly be improved to get a better approximation $\Gamma(pfp_2(\bar{F}))$ of $lfp(F)$. For that purpose, one uses a *truncated descending sequence for an auxiliary operator* $\downarrow\bar{F}$. The initial term in the truncated descending sequence is $pfp_1(\bar{F})$ and the limit of the sequence is $pfp_2(\bar{F})$. The auxiliary functions $\uparrow\bar{F}$ and $\downarrow\bar{F}$ are used in place of $\bar{F}$ to ensure convergence. The schema is the following :



HYPOTHESIS 4.3.2.1 *Let* $\uparrow\bar{F}:\bar{L}^n\to\bar{L}^n$ *and* $S^0\in\bar{L}^n$ *be such that :* $\forall k\geq 0$,

(a) - *if* $\{S^k=\uparrow\bar{F}^k(S^0)$ and not $(\bar{F}(S^k)\sqsubseteq S^k)\}$ *then* $\{(S^k\sqsubseteq\uparrow\bar{F}(S^k))$ and $(\bar{F}(S^k)\sqsubseteq\uparrow\bar{F}(S^k))\}$

(b) - *any strictly ascending chain of the form* $S^0, \uparrow\bar{F}(S^0),\ldots, \uparrow\bar{F}^k(S^0),\ldots$ *is finite.*

DEFINITION 4.3.2.2 *An* ascending strengthened sequence *with initial term* S^0 *is recursively defined by :* $\{S^{k+1}=\uparrow\bar{F}(S^k)$ *iff* not $(\bar{F}(S^k)\sqsubseteq S^k)\}$.

THEOREM 4.3.2.3 *An ascending strengthened sequence is finite ; its limit* $pfp_1(\bar{F})$ *is such that* $lfp(F) \sqsubseteq \Gamma(pfp_1(\bar{F}))$.

Proof : Let m be the (eventually infinite) length of the ascending strengthened sequence. By definition 4.3.2.2 we know that $\forall k<m$ we have not $(\bar{F}(S^k) \sqsubseteq S^k)$ and $S^k = \uparrow\bar{F}^k(S^0)$. Hence by hypothesis 4.3.2.1.a $S^k \sqsubseteq \uparrow\bar{F}(S^k) = S^{k+1}$. Now not $(\bar{F}(S^k) \sqsubseteq S^k)$ either implies that $S^k \sqsubset \bar{F}(S^k)$ in which case $\bar{F}(S^k) \sqsubseteq \uparrow\bar{F}(S^k)$ implies by transitivity $S^k \sqsubset S^{k+1}$, or else $\bar{F}(S^k)$ and S^k are not comparable. But again $S^k = S^{k+1}$ is impossible since otherwise we would have $\bar{F}(S^k) \sqsubseteq \uparrow\bar{F}(S^k) = S^{k+1} = S^k$.

Since $\forall k<m$, $S^k \sqsubset S^{k+1}$, the ascending strengthened sequence form a strictly ascending chain, according to hypothesis 4.3.2.1.b it must be finite.

By definition 4.3.2.2 the last term $S^m = pfp_1(\bar{F})$ of this sequence of successive approximations is such that $\bar{F}(pfp_1(\bar{F})) \sqsubseteq pfp_1(\bar{F})$, hence $\Gamma(pfp_1(\bar{F}))$ correctly approximates $lfp(F)$.
End of Proof.

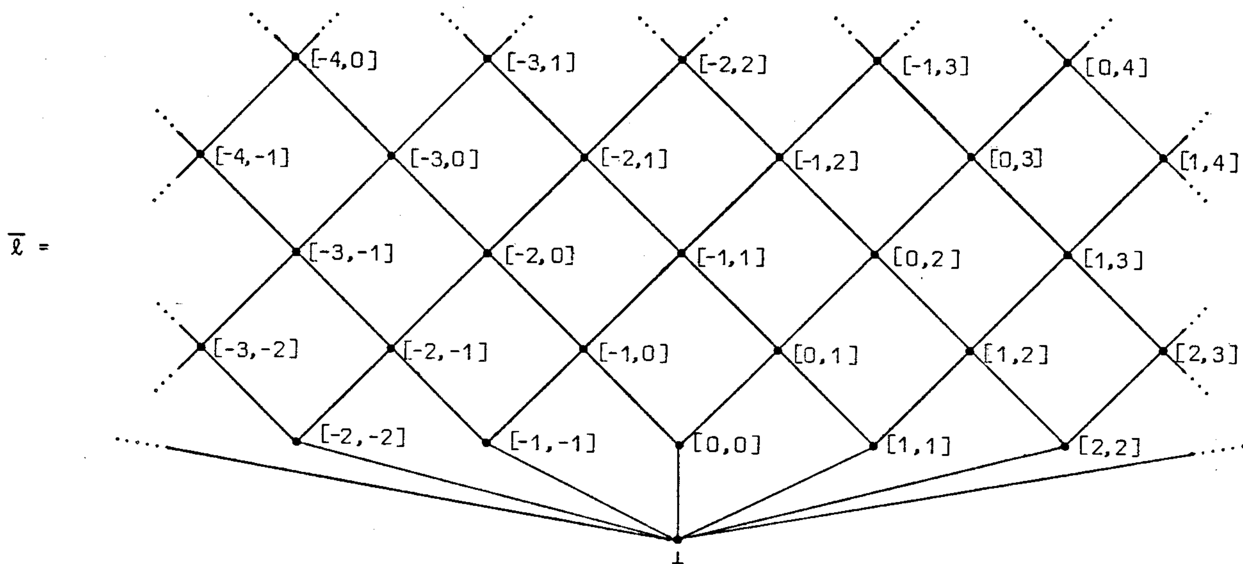
Note : Suppose $\bar{L}$ satisfies the ascending chain condition and $\bar{F}$ is monotone, then one can choose $\uparrow\bar{F}$ to be $\bar{F}$ and S^0 such that $S^0 \sqsubseteq \bar{F}(S^0)$ in which case the ascending strengthened sequence is merely the sequence of successive approximations 2.4.4. Hypothesis 4.3.2.1.b is then equivalent to continuity. *End of Note.*

Example : The analysis of the program :

{1} x:=0; {2} <u>while</u> x≤n <u>do</u> {3} x:=x+2 {4} <u>od</u>; {5}

can be done using an abstract space of properties $\bar{L} = \bar{\ell}^2$, where $\bar{\ell}$ is the following infinite lattice of integer intervals :

$\bar{\ell}$ =



In order to make $\bar{\ell}$ a complete lattice if suffices to allow infinite interval bounds $-\infty$ and $+\infty$. The interpretation is : $\Gamma(\langle\bar{x},\bar{n}\rangle) = \{\langle x,n\rangle \mid x \in \gamma(\bar{x})$ <u>and</u> $n \in \gamma(\bar{n})\}$ where $\gamma(\perp) = \{\Omega\}$ and $\gamma([a,b]) = \{i \mid a \leq i \leq b\} \cup \{\Omega\}$.

The system of equations is :

$$
\begin{cases}
P_1 = \langle \perp, [\alpha,\beta] \rangle \\
P_2 = \langle [0,0],\ P_1.n \rangle \\
P_3' = P_2 \,\bar{\sqcup}\, P_4 = \langle P_2.x \,\bar{\sqcup}\, P_4.x,\ P_2.n \,\bar{\sqcup}\, P_4.n \rangle \\
P_3 = \langle P_3'.x \,\bar{\sqcap}\, [-\infty, \underline{ub}(P_3'.n)],\ [\underline{\ell b}(P_3'.x), +\infty] \,\bar{\sqcap}\, P_3'.n \rangle \\
P_4 = \langle P_3'.x + [2,2],\ P_3.n \rangle \\
P_5 = \langle P_3'.x \,\bar{\sqcap}\, [\underline{\ell b}(P_3'.n)+1, +\infty],\ [-\infty, \underline{ub}(P_3'.x)-1] \,\bar{\sqcap}\, P_3'.n \rangle
\end{cases}
$$

These equations use the union $\bar{\sqcup}$, intersection $\bar{\sqcap}$ and $\underline{ub}$, $\underline{\ell b}$ and + defined on intervals by :

$[a,b]\bar{\sqcup}[c,d] = [\underline{min}(a,c),\underline{max}(b,d)]$

$[a,b]\bar{\sqcap}[c,d] = \underline{if}\ \underline{max}(a,c) \leq \underline{min}(b,d)\ \underline{then}\ [\underline{max}(a,c),\underline{min}(b,d)]\ \underline{else}\ \perp\ \underline{fi}$

$\underline{\ell b}([a,b])=a$; $\underline{ub}([a,b])=b$

$[a,b]+[c,d] = [a+c,b+d]$

The only non obvious equations are P_3 and P_5 taking account of the test $x \leq n$. Suppose $a \leq x \leq b$ and $c \leq n \leq d$. If $x \leq n$ is true, then necessarily $x \leq d$ so that $x \in ([a,b]\bar{\sqcap}[-\infty,d])$ and similarly $a \leq n$ so that $n \in ([c,d]\bar{\sqcap}[a,+\infty])$. Alike, if $x>n$ is true then necessarily $c<x$ so that $x \in ([a,b]\bar{\sqcap}[c+1,+\infty])$ and also $b>n$ so that $n \in ([c,d]\bar{\sqcap}[-\infty,b-1])$.

The initial interval $[\alpha,\beta]$ of n may be given by an input predicate such as a formatted input, or by a type declaration $n \in [\alpha,\beta]$ which has to be run-time checked when the initial value of n is read. A major simplification such as in PASCAL[49], consists in authorizing only manifest constants for bounds declaration, afterwards this will avoid symbolic computations. We will for example take $\alpha=-\infty$, $\beta=1000$.

The least solution (for ordering $\{[a,b]\bar{\sqsubseteq}[c,d]\} \iff \{c \leq a \leq b \leq d\}$) to these equations is :

$$\begin{cases} P_1 = \langle\perp,[-\infty,1000]\rangle \\ P_2 = \langle[0,0],[-\infty,1000]\rangle \\ P_3' = \langle[0,1002],[-\infty,1000]\rangle \\ P_3 = \langle[0,1000],[0,1000]\rangle \\ P_4 = \langle[2,1002],[0,1000]\rangle \\ P_5 = \langle[0,1002],[-\infty,1000]\rangle \end{cases}$$

It is obvious that $\bar{L}$ being an infinite lattice iterative methods are not guaranteed to discover this least solution in a finite number of steps. Consider for example the approximation sequence for solving the equation $x = [1,1]\bar{\sqcup}(x+[1,1])$. It is an infinite sequence the first terms of which are $\perp$, $[1,1]$, $[1,2]$, $[1,3]$,... and the limit of which is $[1,-\infty]$.

Since a compiler must not enter an endless cycle, it must approximate the limits of potentially infinite approximation sequences. For that purpose one can use heuristics which induce an approximation of the expected limit from the first few terms of the sequence. The simplest heuristic which can be used with intervals is probably the following : if a bound of an interval is not constant take it to be infinite.

To put in practice this heuristic, let us introduce the *widening* ∇ of intervals. The empty interval $\perp$ is the null element of ∇, and otherwise

$[a,b]\nabla[c,d] = [\underline{if}\ c<a\ \underline{then}\ -\infty\ \underline{else}\ a\ \underline{fi},\ \underline{if}\ d>b\ \underline{then}\ +\infty\ \underline{else}\ b\ \underline{fi}]$

The strengthened system of equations $\bar{P}=\dagger\bar{F}(\bar{P})$ is obtained by modification of equation P_3' (corresponding to a loop head) :

$P_3' = \langle P_3'.x\ \nabla\ (P_2.x\bar{\sqcup}P_4.x),\ P_3'.n\ \nabla\ (P_2.n\bar{\sqcup}P_4.n)\rangle$

Note that $\dagger\bar{F}$ is not monotone since for example $([0,1]\bar{\sqsubseteq}[0,4])$ and $([0,2]\bar{\sqsubseteq}[0,3])$ does not imply that $[0,1]\nabla[0,2]=[0,+\infty]$ is less than or equal to $[0,4]\nabla[0,3]=[0,4]$. Hypothesis 4.3.2.1.a is clearly satisfied since $([a,b]\bar{\sqcup}[c,d])\ \bar{\sqsubseteq}\ ([a,b]\nabla[c,d])$.

The ascending strengthened sequence is now :

Initialization :

$$\left[P_1^0=P_2^0=P_3'^0=P_3^0=P_4^0=P_5^0=\langle\perp,\perp\rangle \right.$$

Step 1 :

$$
\left[\begin{array}{lll}
P_1^1 & = & \langle \bot, [-\infty, 1000] \rangle \\
P_2^1 & = & \langle [0,0], P_1^1.n \rangle \\
 & = & \langle [0,0], [-\infty, 1000] \rangle \\
P'^1_3 & = & \langle P'^0_3.x \,\nabla\, (P_2^1.x \,\overline{\sqcup}\, P_4^0.x),\ P'^0_3.n \,\nabla\, (P_2^1.n \,\overline{\sqcup}\, P_4^1.n) \rangle \\
 & = & \langle \bot \,\nabla\, ([0,0] \,\overline{\sqcup}\, \bot),\ \bot \,\nabla\, ([-\infty, 1000] \,\overline{\sqcup}\, \bot) \rangle \\
 & = & \langle \bot \,\nabla\, [0,0],\ \bot \,\nabla\, [-\infty, 1000] \rangle \\
 & = & \langle [0,0], [-\infty, 1000] \rangle \\
P_3^1 & = & \langle P'^1_3.x \,\overline{\sqcap}\, [-\infty, \underline{ub}(P'^1_3.n)],\ [\underline{\ell b}(P'^1_3.x), +\infty] \,\overline{\sqcap}\, P'^1_3.n \rangle \\
 & = & \langle [0,0] \,\overline{\sqcap}\, [-\infty, 1000],\ [0, +\infty] \,\overline{\sqcap}\, [-\infty, 1000] \rangle \\
 & = & \langle [0,0], [0, 1000] \rangle \\
P_4^1 & = & \langle P_3^1.x + [2,2],\ P_3^1.n \rangle \\
 & = & \langle [2,2], [0, 1000] \rangle \\
P_5^1 & = & \langle P'^1_3.x \,\overline{\sqcap}\, [\underline{\ell b}(P'^1_3.n)+1, +\infty],\ [-\infty, \underline{ub}(P'^1_3.x)-1] \,\overline{\sqcap}\, P'^1_3.n \rangle \\
 & = & \langle [0,0] \,\overline{\sqcap}\, [-\infty, +\infty],\ [-\infty, -1] \,\overline{\sqcap}\, [-\infty, 1000] \rangle \\
 & = & \langle [0,0], [-\infty, -1] \rangle
\end{array}\right.
$$

Since $\underline{\text{not}}\ ((P_4^1 \,\overline{\sqcup}\, P_2^1) \,\overline{\sqsubseteq}\, P'^1_3)$ we have to go on step 2 :

$$
\left[\begin{array}{lll}
P_2^2 & = & \langle [0,0], [-\infty, 1000] \rangle \\
P'^2_3 & = & \langle [0,0] \,\nabla\, ([0,0] \,\overline{\sqcup}\, [2,2]),\ [-\infty, 1000] \,\nabla\, ([-\infty, 1000] \,\overline{\sqcup}\, [0, 1000]) \rangle \\
 & = & \langle [0, +\infty], [-\infty, 1000] \rangle \\
P_3^2 & = & \langle [0, +\infty] \,\overline{\sqcap}\, [-\infty, 1000],\ [0, +\infty] \,\overline{\sqcap}\, [-\infty, 1000] \rangle \\
 & = & \langle [0, 1000], [0, 1000] \rangle \\
P_4^2 & = & \langle [2, 1002], [0, 1000] \rangle \\
P_5^2 & = & \langle [0, +\infty] \,\overline{\sqcap}\, [-\infty, +\infty], [-\infty, +\infty] \,\overline{\sqcap}\, [-\infty, 1000] \rangle \\
 & = & \langle [0, +\infty], [-\infty, 1000] \rangle
\end{array}\right.
$$

Notice that we have found $pfp_1(\overline{F})$ since $\overline{F}(P) \,\overline{\sqsubseteq}\, P$ because of $((P_4^2 \,\overline{\sqcup}\, P_2^2) \,\overline{\sqsubseteq}\, P'^2_3)$ that is $\langle [0,1002], [-\infty, 1000] \rangle \,\overline{\sqsubseteq}\, \langle [0, +\infty], [-\infty, 1000] \rangle$. The approximate solution is therefore :

$$
\left[\begin{array}{lll}
P_1 & = & \langle \bot, [-\infty, 1000] \rangle \\
P_2 & = & \langle [0,0], [-\infty, 1000] \rangle \\
P'_3 & = & \langle [0, +\infty], [-\infty, 1000] \rangle \\
P_3 & = & \langle [0, 1000], [0, 1000] \rangle \\
P_4 & = & \langle [2, 1002], [0, 1000] \rangle \\
P_5 & = & \langle [0, +\infty], [-\infty, 1000] \rangle
\end{array}\right.
$$

Termination is always guaranteed since widenings take place at least once along each cycle in the graph of dependence of the system of equations and forbid infinite strictly increasing chains by passing to infinite bounds [12]. *End of Example.*

In case $\overline{F}$ is monotone the limit $pfp_1(\overline{F})$ of the ascending strengthened sequence can be chosen to be the initial term of a descending approximation sequence. In fact $\overline{F}(pfp_1(\overline{F})) \,\overline{\sqsubseteq}\, pfp_1(\overline{F})$ and $lfp(F) \sqsubseteq \Gamma(pfp_1(\overline{F}))$ imply $lfp(F) \sqsubseteq \Gamma(\overline{F}^k(pfp_1(\overline{F}))) \sqsubseteq \Gamma(pfp_1(\overline{F}))$, $\forall k \geq 0$. Therefore if the limit $pfp_2(\overline{F}) = \underline{\lim}_{k \to \infty} \overline{F}^k(pfp_1(\overline{F}))$ exists, it is a better approximation of $lfp(F)$ than $pfp_1(\overline{F})$ (as soon as $pfp_1(\overline{F})$ is not a fixed point of $\overline{F}$).

Again when this descending sequence is infinite or in practice slowly converging we can approximate its limit. Note however that dualizing the ascending strengthened sequence would not be convenient, since it would lead to a lower approximation of $lfp(F)$ whereas an upper approximation is desired. We must have $lfp(F) \,\overline{\sqsubseteq}\, \Gamma(pfp_2(\overline{F}))$ since we want to take account of all states which can occur during any program execution.

HYPOTHESIS 4.3.2.4 — $\bar{F}:\bar{L}^n \rightarrow \bar{L}^n$ *is monotone,*

— *let* $\downarrow\bar{F}:\bar{L}^n \rightarrow \bar{L}^n$ *be such that :*

- $\forall S \in \bar{L}^n$, $\{\bar{F}(S) \sqsubset S\} \Longrightarrow \{\bar{F}(S) \sqsubseteq \downarrow\bar{F}(S) \sqsubseteq S\}$
- *every strictly descending chain of the form* $S^0 \in \bar{L}^n$, $\downarrow\bar{F}(S^0), \ldots, \downarrow\bar{F}^k(S^0)$, ... *is finite.*

DEFINITION 4.3.2.5 *A* truncated descending sequence *with initial term* $S^0 \in \bar{L}^n$ *such that* $lfp(F) \subseteq \Gamma(\bar{F}(S^0))$ and $\bar{F}(S^0) \sqsubseteq S^0$ *is recursively defined by :* $S^{k+1} = \downarrow\bar{F}(S^k)$ *iff* $S^k \neq \bar{F}(S^k)$ and $S^k \neq \downarrow\bar{F}(S^k)$.

THEOREM 4.3.2.6 *A truncated descending sequence is finite, its limit* $pfp_2(\bar{F})$ *is such that* $lfp(F) \subseteq \Gamma(pfp_2(\bar{F}))$.

Proof : Let p+1 be the (eventually infinite) length of the truncated descending sequence. Assuming $p \geq 0$, we prove by recurrence on k that :

$$\bar{F}(S^{k-1}) \sqsubseteq S^k \sqsubset S^{k-1} \text{ and } lfp(F) \subseteq \Gamma(S^k),\ \forall k \leq p$$

Basis : Since $\bar{F}(S^0) \sqsubseteq S^0$ and $S^0 \neq \bar{F}(S^0)$ we have $\bar{F}(S^0) \sqsubset S^0$. Hence hypothesis 4.3.2.4 implies $\bar{F}(S^0) \sqsubseteq \downarrow\bar{F}(S^0) \sqsubseteq S^0$, moreover $S^0 \neq \downarrow\bar{F}(S^0)$ so that by definition 4.3.2.5 we have $\bar{F}(S^0) \sqsubseteq \downarrow\bar{F}(S^0) = S^1 \sqsubset S^0$. Besides, $lfp(F) \subseteq \Gamma(\bar{F}(S^0)) \subseteq \Gamma(S^1)$ by monotony of Γ.

Induction hypothesis : Let us suppose that $\forall k<p$ we have :

$$\bar{F}(S^{k-1}) \sqsubseteq S^k \sqsubset S^{k-1} \text{ and } lfp(F) \subseteq \Gamma(S^k)$$

we prove this is also true for k+1. Since $\bar{F}$ is monotone we have by transitivity $\bar{F}(S^k) \sqsubseteq S^k$ and since k<p definition 4.3.2.5 implies $\bar{F}(S^k) \neq S^k$, hence $\bar{F}(S^k) \sqsubset S^k$. According to hypothesis 4.3.2.4 $\bar{F}(S^k) \sqsubseteq \downarrow\bar{F}(S^k) \sqsubseteq S^k$, but again since k<p we have $\downarrow\bar{F}(S^k) \neq S^k$ and $\bar{F}(S^k) \sqsubseteq \downarrow\bar{F}(S^k) = S^{k+1} \sqsubset S^k$. Now $lfp(F) \subseteq \Gamma(S^k)$ thus $lfp(F) \subseteq F(\Gamma(S^k))$ since F is order-preserving. Hypothesis 4.3.0.4.b then implies that $lfp(F) \subseteq \Gamma(\bar{F}(S^k))$ thus $lfp(F) \subseteq \Gamma(S^{k+1})$ by monotony of Γ and transitivity.

Finally by recurrence on k, the truncated descending sequence is a strictly descending chain. Then hypothesis 4.3.2.4 implies that p is finite. Moreover for k=p we have $lfp(F) \subseteq \Gamma(S^p) = \Gamma(pfp_2(\bar{F}))$. *End of Proof.*

Example : The truncated descending sequence can be defined by taking $\downarrow\bar{F} = \bar{F}$ and imposing an arbitrary upper bound on its length. However it is better to stop the iteration process when the approximation is found to be precise enough. For example, we can try eliminating infinite bounds of intervals using a truncated descending sequence and stop iterating when no more infinite bound can be eliminated.

For that purpose let us introduce the *narrowing* Δ of intervals. The empty interval $\perp$ is the null element of Δ, and otherwise $[a,b] \Delta [c,d]$ = [if a=$-\infty$ then c else min(a,c) fi, if b=$+\infty$ then d else max(b,d) fi].

The system of equations $\bar{P} = F(\bar{P})$ is modified into $\bar{P} = \downarrow\bar{F}(\bar{P})$ by modification of equation P_3' (corresponding to a loop head) :

$$P_3' = \langle P_3'.x \,\Delta\, (P_2.x \,\bar{\sqcup}\, P_4.x),\ P_3'.n \,\Delta\, (P_2.n \,\bar{\sqcup}\, P_4.n) \rangle$$

The truncated descending sequence is initialized with $pfp_1(\bar{F})$:

Initialization :

$$\begin{cases} P_1^0 = \langle \perp, [-\infty, 1000] \rangle \\ P_2^0 = \langle [0,0], [-\infty, 1000] \rangle \\ P_3'^0 = \langle [0,+\infty], [-\infty, 1000] \rangle \\ P_3^0 = \langle [0,1000], [0,1000] \rangle \\ P_4^0 = \langle [2,1002], [0,1000] \rangle \\ P_5^0 = \langle [0,+\infty], [-\infty, 1000] \rangle \end{cases}$$

Step 1 :

$$
\begin{aligned}
P_1^1 &= \langle \bot, [-\infty, 1000] \rangle \\
P_2^1 &= \langle [0,0], [-\infty, 1000] \rangle \\
P_3'^1 &= \langle P_3'^0.x \,\Delta\, (P_2^1.x \,\bar{\sqcup}\, P_4^0.x),\ P_3'^0.n \,\Delta\, (P_2^1.n \,\bar{\sqcup}\, P_4^0.n) \rangle \\
&= \langle [0,+\infty] \,\Delta\, ([0,0] \,\bar{\sqcup}\, [2,1002]),\ [-\infty,1000] \,\Delta\, ([-\infty,1000] \,\bar{\sqcup}\, [0,1000]) \rangle \\
&= \langle [0,+\infty] \,\Delta\, [0,1002],\ [-\infty,1000] \,\Delta\, [-\infty,1000] \rangle \\
&= \langle [0,1002], [-\infty,1000] \rangle \\
P_3^1 &= \langle P_3'^1.x \,\bar{\sqcap}\, [-\infty, \underline{ub}(P_3'^1.n)],\ [\underline{\ell b}(P_3'^1.x), +\infty] \,\bar{\sqcap}\, P_3'^1.n \rangle \\
&= \langle [0,1002] \,\bar{\sqcap}\, [-\infty,1000],\ [0,+\infty] \,\bar{\sqcap}\, [-\infty,1000] \rangle \\
&= \langle [0,1000], [0,1000] \rangle \\
P_4^1 &= \langle P_3^1.x + [2,2],\ P_3^1.n \rangle \\
&= \langle [2,1002], [0,1000] \rangle \\
P_5^1 &= \langle P_3'^1.x \,\bar{\sqcap}\, [\underline{\ell b}(P_3'^1.n)+1, +\infty],\ [-\infty, \underline{ub}(P_3'^1.x)-1] \,\bar{\sqcap}\, P_3'^1.n \rangle \\
&= \langle [0,1002] \,\bar{\sqcap}\, [-\infty,+\infty],\ [-\infty,1001] \,\bar{\sqcap}\, [-\infty,1000] \rangle \\
&= \langle [0,1002], [-\infty,1000] \rangle
\end{aligned}
$$

Now it is easy to verify that $P_4^1 \bar{\sqcup} P_2^1 = P_3'^1$ so that $\bar{F}(P)=P$, the truncated decreasing sequence has converged to a fixed point of $\bar{F}$. It happens that it is the least fixed point of $\bar{F}$ but in general this is not necessarily the case. It is important to note that because of the undecidable problems compilers are faced with, the approximation of infinite iterations is valid but fundamentally incomplete. However, it should be clear that this incompleteness is acceptable to compilers which never need full knowledge of the properties of the programs. *End of Example.*

4.4 VERIFICATION OF THE CORRECTNESS OF AN EXACT OR APPROXIMATE SOLUTION

Finally, when the solution S to the equations X=F(X) cannot be automatically computed, it may be provided by the programmer. Yet, S must be verified to be correct.

4.4.1 VERIFICATION OF THE CORRECTNESS OF THE EXACT SOLUTION

The least fixed point S of F must be provided by giving the value S(k) of the general term S^k of the ascending approximation sequence $S^0, \ldots, S^k, \ldots$. According to 4.2 the problem is then to verify that $\exists i \geq 0$ such that $\{(S(i) \subseteq F(S(i)))$ <u>and</u> $(S(i) \subseteq lfp(F))\}$ and $\forall k \geq i$, $S(k+1)=F(S(k))$. The solution S to the equations is obtained by passing to the limit $S = \lim_{k \to \infty}(S(k))$.

Note that according to theorem 2.5.4 the above verification rule can be extended to periodic chaotic iterations, when the period length is bounded.

4.4.2 VERIFICATION OF THE CORRECTNESS OF APPROXIMATE SOLUTIONS

In practice it may be impossible for the programmer to guess the exact solution S, although he may be able to provide an approximate solution $\bar{S}$. The problem is to verify that $S \sqsubseteq \bar{S}$ in spite of the fact that S is unknown. But according to theorem 4.3.0.2, it is sufficient to verify that $F(\bar{S}) \sqsubseteq \bar{S}$. This verification rule was called fixed point induction by Park[48].

5. APPLICATION TO THE DENOTATIONAL SEMANTICS OF PROGRAMMING LANGUAGES

Mathematical or denotational semantics was introduced by Scott and Strachey ([52],[54]) and further developed by several authors. The complete details and a guide to the literature may be found in [53].

5.1 FUNCTIONS

Suppose that each program variable takes its values in a domain D including some special value Ω which is the value of uninitialized variables.

If the program has n variables, we shall consider the state space D^n, and denote $D^{n'}=D^n \cup \{\bot,\top\}$. $D^{n'}$ is made a complete lattice using the ordering $\sqsubseteq_{D^{n'}}$ defined by :

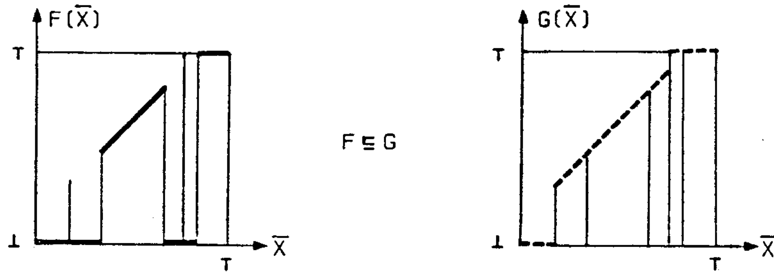
$\bot \sqsubseteq_{D^{n'}} \bot \sqsubseteq_{D^{n'}} \bar{X} \sqsubseteq_{D^{n'}} \bar{X} \sqsubseteq_{D^{n'}} \top \sqsubseteq_{D^{n'}} \top$, $\forall \bar{X} \in D^n$.

The semantics of a program P is the partial function $F_P : D^n \to D^n$ computed by that program. Therefore if the initial values of the program variables are $\bar{X}_0$, their final values will be $F_P(\bar{X}_0)$ after execution of the program.

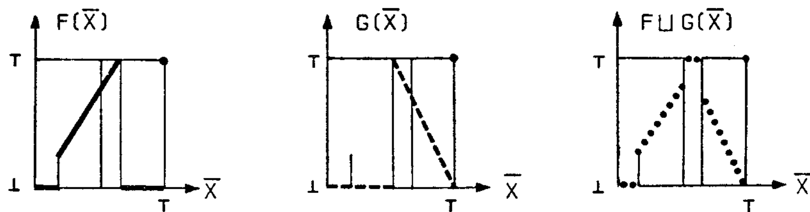
As usual a partial function $F:D^n \to D^n$ is considered to be a total function $F:D^n \to D^{n'}$ such that $F(\bar{X})=\bot$ whenever $F(\bar{X})$ is undefined for $\bar{X} \in D^n$. Moreover we naturally extend F to $D^{n'} \to D^{n'}$ by defining $F(\bot)=\bot$ and $F(\top)=\top$.

Let us now define an ordering $\sqsubseteq$ for functions in $D^{n'} \to D^n$ by : $\{F \sqsubseteq G\} \iff \{\forall \bar{X} \in D^{n'}, F(\bar{X}) \sqsubseteq_{D^{n'}} G(\bar{X})\}$ and let $\sqcup$ be the corresponding least upper bound operation.

In order to visualize these operations, consider a "projection" of D^n on the real open interval]0,1[. Take $\bot=0$ and $\top=1$. The "projection" of the ordering $\sqsubseteq_{D^{n'}}$ on [0,1] is not the usual $\leq$ since any two distinct points of D^n are not comparable.
An example of comparable functions F and G would be



An example of union of (non-comparable) functions F and G would be :



5.2 FUNCTIONAL EQUATIONS

Let us now define the syntactic mechanism which permits associating a system of equations with any sequential program.

Go to statements and labels : Let L be a constant label ...$\{F_j\}$;L:$\{F\}$.:. which can be reached sequentially or by unconditional jump statements $\{F_i\}$ go to L;. The function computed after the label L is $F = \coprod_{k \in \underline{pred}(L)} F_k$ where pred(L) denotes the set of program points going to L

sequentially or by jumps.

Assignment statements : We consider parallel assignments $\overline{X} \leftarrow F(\overline{X})$ of n values $F(\overline{X})$ to n variables. The semantics of $\{F_1\}$ $\overline{X} \leftarrow F(\overline{X})$ $\{F_2\}$ is given by $F_2 = F \circ F_1$ using functional composition denoted by $\circ$.

Conditional statements : The semantics of $\{F\}$ <u>if</u> $P(\overline{X})$ <u>then</u> $\{F_t\}$...<u>else</u>$\{F_f\}$...<u>fi</u> is defined by $F_t = (\iota | P) \circ F$ and $F_f = (\iota | \underline{\text{not}}\ P) \circ F$.
We denote by ι the identity function, that is $\iota(\overline{X}) = \overline{X}$, $\forall \overline{X} \in D^{n'}$. If $F : D^{n'} \to D^{n'}$ is a function and P a predicate $(P : D^{n'} \to \{\underline{\text{true}}, \underline{\text{false}}\})$, we denote by $(F|P)$ the restriction of the function F to the subset of $D^{n'}$ satisfying the predicate P, therefore : $\forall \overline{X} \in D^{n'}$, $(F|P)(\overline{X})$ = <u>if</u> $P(\overline{X})$ <u>then</u> $F(\overline{X})$ <u>else</u> $\bot$ <u>fi</u>.

Let us now quote some useful properties of the functional operations $\circ$ (composition), $\sqcup$ (union) and $|$ (restriction). We define $\bot_{D^{n'} \to D^{n'}}$ to be the constant function which result is always the infimum of $D^{n'}$. As before we note $(F)^0 = \iota$ and $(F)^{n+1} = F \circ (F)^n$.

$$\bot_{D^{n'} \to D^{n'}} \circ F = F \circ \bot_{D^{n'} \to D^{n'}} = \bot_{D^{n'} \to D^{n'}}$$

$$\bot_{D^{n'} \to D^{n'}} \sqcup F = F \sqcup \bot_{D^{n'} \to D^{n'}} = F$$

$$(F | \underline{\text{true}}) = F$$

$$(F | \underline{\text{false}}) = \bot_{D^{n'} \to D^{n'}}$$

$$(G|P) \circ (F|Q) = (G \circ F | P \circ F\ \underline{\text{and}}\ Q)$$

$$(G|P) \circ F = (G \circ F | P \circ F)$$

$$G \circ (F|Q) = (G \circ F | Q)$$

$$F \circ (\coprod_{i \in \Delta} G_i) = \coprod_{i \in \Delta} (F \circ G_i)$$

5.3 EXAMPLE OF THE SEMANTICS OF WHILE LOOPS

A while loop such as : <u>while</u> $P(\overline{X})$ <u>do</u> $\overline{X} \leftarrow F(\overline{X})$ <u>od</u> ; over the set of variables $\overline{X}$ is a syntactic denotation of the program schema :

{0} loop: {1} <u>if</u> $P(\overline{X})$ <u>then</u> {2} $\overline{X} \leftarrow F(\overline{X})$ {3} <u>go to</u> loop; <u>fi</u>; {4}

Its semantics is given by the least fixed point of the system of equations :

$$\begin{cases} F_0 = \iota \\ F_1 = F_0 \sqcup F_3 \\ F_2 = (\iota | P) \circ F_1 \\ F_3 = F \circ F_2 \\ F_4 = (\iota | \underline{\text{not}}\ P) \circ F_1 \end{cases}$$

The least fixed point is the limit of the ascending approximation sequence (theorem 2.4.4) also called *Kleene's sequence* in this particular application. However according to theorem 2.5.3 we can use the Gauss-Seidel transform of Kleene's sequence :

Initialization :

$$\left[F_i^0 = \bot_{D^{n'} \to D^{n'}},\quad i = 0..4 \right.$$

Step 1 :

$$\left[\begin{array}{l} F_0^1 = \iota \\ F_1^1 = F_0^1 \sqcup F_3^0 = \iota \sqcup \bot_{D^{n'} \to D^{n'}} = \iota \\ F_2^1 = (\iota | P) \circ F_1^1 = (\iota | P) \circ \iota = (\iota | P) \end{array} \right.$$

$$\left\lfloor\begin{array}{l} F_3^1 = F\circ F_2^1 = F\circ(\iota|P) = (F|P) \\ F_4^1 = (\iota|\,\underline{\text{not}}\,P)\circ F_1^1 = (\iota|\,\underline{\text{not}}\,P)\circ\iota = (\iota|\,\underline{\text{not}}\,P) \end{array}\right.$$

Step 2 :

$$\left[\begin{array}{l} F_0^2 = \iota \\ F_1^2 = F_0^2 \sqcup F_3^1 = \iota \sqcup (F|P) \\ F_2^2 = (\iota|P)\circ F_1^2 = (\iota|P)\circ(\iota\sqcup(F|P)) = ((\iota|P)\circ\iota)\sqcup((\iota|P)\circ(F|P)) \\ \quad = (\iota|P)\sqcup(F|P\ \underline{\text{and}}\ P\circ F) \\ F_3^2 = F\circ F_2^2 = F\circ((\iota|P)\sqcup(F|P\ \underline{\text{and}}\ P\circ F)) = (F|P)\sqcup((F)^2|P\ \underline{\text{and}}\ P\circ F) \\ F_4^2 = (\iota|\,\underline{\text{not}}\,P)\circ F_1^2 = (\iota|\,\underline{\text{not}}\,P)\circ(\iota\sqcup(F|P)) = ((\iota|\,\underline{\text{not}}\,P)\circ\iota)\sqcup((\iota|\,\underline{\text{not}}\,P)\circ(F|P)) \\ \quad = (\iota|\,\underline{\text{not}}\,P)\sqcup(F|P\ \underline{\text{and}}\ \underline{\text{not}}\ P\circ F) \end{array}\right.$$

By finding these first few approximations we are led to the formulas :

Step j :

$$\left[\begin{array}{l} F_0^j = \iota \\ F_1^j = \coprod\limits_{k=0}^{j-1}((F)^k|\mathop{\underline{\text{AND}}}\limits_{i=0}^{k-1} P\circ(F)^i) \\ F_2^j = \coprod\limits_{k=0}^{j-1}((F)^k\,|\,\mathop{\underline{\text{AND}}}\limits_{i=0}^{k} P\circ(F)^i) \\ F_3^j = \coprod\limits_{k=0}^{j-1}((F)^{k+1}|\mathop{\underline{\text{AND}}}\limits_{i=0}^{k} P\circ(F)^i) \\ F_4^j = \coprod\limits_{k=0}^{j-1}((F)^k|(\mathop{\underline{\text{AND}}}\limits_{i=0}^{k-1} P\circ(F)^i)\ \underline{\text{and}}\ \underline{\text{not}}\ P\circ(F)^k) \end{array}\right.$$

These may then be proved to be correct using mathematical induction (4.4.1) :

- It is first easy to verify that the above formulas are correct for j=0,1 and 2 with the usual conventions that $\coprod\limits_{i\in\Delta} F_i = \bot_{D^{n'}\to D^{n'}}$ and $\mathop{\underline{\text{AND}}}\limits_{i\in\Delta} P_i = \underline{\text{true}}$ hold when the indexing set Δ is empty.

- Replacing the unknowns in the right hand side of the equations by the hypothetical values of step j we get at step j+1 :

$$\left[\begin{array}{l} F_0^{j+1} = \iota \\ F_1^{j+1} = F_0^{j+1}\sqcup F_3^j = \iota\sqcup(\coprod\limits_{k=0}^{j-1}((F)^{k+1}|\mathop{\underline{\text{AND}}}\limits_{i=0}^{k} P\circ(F)^i)) = \iota\sqcup(\coprod\limits_{k'=1}^{j}((F)^{k'}|\mathop{\underline{\text{AND}}}\limits_{i=0}^{k'-1} P\circ(F)^i)) \\ \quad = \coprod\limits_{k=0}^{(j+1)-1}((F)^k|\mathop{\underline{\text{AND}}}\limits_{i=0}^{k-1} P\circ(F)^i) \\ F_2^{j+1} = (\iota|P)\circ F_1^{j+1} = (\iota|P)\circ(\coprod\limits_{k=0}^{j}((F)^k|\mathop{\underline{\text{AND}}}\limits_{i=0}^{k-1} P\circ(F)^i)) = \coprod\limits_{k=0}^{j}(\iota|P)\circ((F)^k|\mathop{\underline{\text{AND}}}\limits_{i=0}^{k-1} P\circ(F)^i) \\ \quad = \coprod\limits_{k=0}^{j}(\iota\circ(F)^k\,|\,P\circ(F)^k\ \underline{\text{and}}\ (\mathop{\underline{\text{AND}}}\limits_{i=0}^{k-1} P\circ(F)^i)) = \coprod\limits_{k=0}^{(j+1)-1}((F)^k|\mathop{\underline{\text{AND}}}\limits_{i=0}^{k} P\circ(F)^i) \\ F_3^{j+1} = F\circ F_2^{j+1} = F\circ(\coprod\limits_{k=0}^{j}((F)^k|\mathop{\underline{\text{AND}}}\limits_{i=0}^{k} P\circ(F)^i)) = \coprod\limits_{k=0}^{j}(F\circ((F)^k|\mathop{\underline{\text{AND}}}\limits_{i=0}^{k} P\circ(F)^i)) \\ \quad = \coprod\limits_{k=0}^{(j+1)-1}((F)^{k+1}|\mathop{\underline{\text{AND}}}\limits_{i=0}^{k} P\circ(F)^i) \\ F_4^{j+1} = (\iota|\,\underline{\text{not}}\,P)\circ F_1^{j+1} = (\iota|\,\underline{\text{not}}\,P)\circ(\coprod\limits_{k=0}^{j}((F)^k|\mathop{\underline{\text{AND}}}\limits_{i=0}^{k-1} P\circ(F)^i)) \\ \quad = \coprod\limits_{k=0}^{j}(\iota|\,\underline{\text{not}}\,P)\circ((F)^k|\mathop{\underline{\text{AND}}}\limits_{i=0}^{k-1} P\circ(F)^i) = \coprod\limits_{k=0}^{j}(\iota\circ(F)^k|(\underline{\text{not}}\ P\circ(F)^k)\ \underline{\text{and}}\ (\mathop{\underline{\text{AND}}}\limits_{i=0}^{k-1} P\circ(F)^i)) \\ \quad = \coprod\limits_{k=0}^{(j+1)-1}((F)^k|(\mathop{\underline{\text{AND}}}\limits_{i=0}^{k-1} P\circ(F)^i)\ \underline{\text{and}}\ (\underline{\text{not}}\ P\circ(F)^k)) \end{array}\right.$$

The general term of the (Gauss-Seidel transform of) Kleene's sequence given at step j has been proved to be correct by recurrence on j. The limit of Kleene's sequence is obtained when $j\to\infty$, so that the function computed by the while-loop schema is :

$$\lim_{j\to\infty} F_4^j = \coprod_{k=0}^{\infty} ((F)^k \mid (\underline{\text{AND}}_{i=0}^{k-1} P\circ(F)^i) \ \underline{\text{and}}\ (\underline{\text{not}}\ P\circ(F)^k))$$

5.4 APPLICATION TO A PROGRAM FOR COMPUTING $\lfloor\sqrt{a}\rfloor$

Let us consider the following program (taken in [44]) :

```
<x,y,z> ← <0,1,1>;
while y≤a do
    <x,y,z> ← <x+1,y+z+2,z+2>;
od;
```

It computes the integer square root $\lfloor\sqrt{a}\rfloor$ of a natural integer "a" using the arithmetic property: $\forall n\in\mathbb{N},\ 1+3+\ldots+(2n-1)=n^2$. The semantics of this program is the function : $R=\underline{\text{while}}\circ\mu$ where :

$$\mu(<x,y,z>) = <0,1,1>$$

$$\underline{\text{while}} = \coprod_{k=0}^{\infty} ((F)^k \mid (\underline{\text{AND}}_{i=0}^{k-1} P\circ(F)^i) \ \underline{\text{and}}\ (\underline{\text{not}}\ P\circ(F)^k))$$

with $P(<x,y,z>) = (y\le a)$ and $F(<x,y,z>) = <x+1,y+z+2,\ z+2>$. We have :

$$R = \underline{\text{while}}\circ\mu = (\coprod_{k=0}^{\infty} ((F)^k \mid (\underline{\text{AND}}_{i=0}^{k-1} P\circ(F)^i) \ \underline{\text{and}}\ (\underline{\text{not}}\ P\circ(F)^k)))\circ\mu$$

$$= \coprod_{k=0}^{\infty} (((F)^k \mid (\underline{\text{AND}}_{i=0}^{k-1} P\circ(F)^i) \ \underline{\text{and}}\ (\underline{\text{not}}\ P\circ(F)^k))\circ\mu)$$

$$= \coprod_{k=0}^{\infty} ((F)^k\circ\mu \mid (\underline{\text{AND}}_{i=0}^{k-1} P\circ(F)^i\circ\mu) \ \underline{\text{and}}\ (\underline{\text{not}}\ P\circ(F)^k\circ\mu))$$

Let us compute $(F)^n\circ\mu$ for $n\ge 0$:

$(F)^0\circ\mu=1\circ\mu=\mu=<0,1,1>$. Induction hypothesis : $(F)^j\circ\mu = <j,(j+1)^2,2j+1>$. Then :

$(F)^{j+1}\circ\mu = <j+1,(j+1)^2+2j+1+2,2j+1+2> = <j+1,((j+1)+1)^2,2(j+1)+1>$. By recurrence :

$(F)^n\circ\mu = <n,(n+1)^2,2n+1>$, $\forall n\ge 0$ and also $P\circ(F)^n\circ\mu = ((n+1)^2\le a)$.

Substituting in R we get :

$$R = \coprod_{k=0}^{\infty} (<k,(k+1)^2,2k+1> \mid (\underline{\text{AND}}_{i=0}^{k-1}(i+1)^2 \le a) \ \underline{\text{and}}\ \underline{\text{not}}\ ((k+1)^2\le a))$$

Simplifying using the arithmetic property :

$$(\underline{\text{AND}}_{i=0}^{k-1}(i+1)^2\le a) \iff (\underline{\text{AND}}_{i=1}^{k} i^2\le a) \iff (k^2\le a)$$

We obtain : $R = \coprod_{k=0}^{\infty} (<k,(k+1)^2,2k+1> \mid k^2\le a<(k+1)^2)$.

Note that the predicate $k^2\le a<(k+1)^2$ is true only for a unique value $\lfloor\sqrt{a}\rfloor$ of k, therefore R simplifies to :

$$R = (\coprod_{k\ne\lfloor\sqrt{a}\rfloor} (<k,(k+1)^2,2k+1> \mid \underline{\text{false}})) \sqcup (<\lfloor\sqrt{a}\rfloor,(\lfloor\sqrt{a}\rfloor+1)^2,2\lfloor\sqrt{a}\rfloor+1> \mid \underline{\text{true}})$$

$$= (\coprod_{k\ne\lfloor\sqrt{a}\rfloor} \bot_{D^{n'}\to D^{n'}}) \sqcup (<\lfloor\sqrt{a}\rfloor,(\lfloor\sqrt{a}\rfloor+1)^2,2\lfloor\sqrt{a}\rfloor+1>)$$

$$= <\lfloor\sqrt{a}\rfloor,(\lfloor\sqrt{a}\rfloor+1)^2,2\lfloor\sqrt{a}\rfloor+1>$$

which is the expected result of the program.

5.5 REMARKS

Remark 1 : implicit or explicit semantics of commands.

There are two ways of expressing the semantics of the while command : while $P(\overline{X})$ do $\overline{X} \leftarrow F(\overline{X})$ od;

1. *"static" or implicit definition :* The function computed by the while command is the term $F_4(\infty)$ of the least solution $\langle F_1(\infty), F_4(\infty)\rangle$ to the functional equations :

$$\begin{cases} F_1 = \iota \sqcup ((F|P) \circ F_1) \\ F_4 = (\iota | \underline{\text{not}}\, P) \circ F_1 \end{cases}$$

2. *"dynamic" or explicit definition :* The function computed by the while command is :

$$\coprod_{k=0}^{\infty} ((F)^k | (\underset{i=0}{\overset{k-1}{\text{AND}}} P \circ (F)^i) \ \underline{\text{and}}\ (\underline{\text{not}}\ P \circ (F)^k))$$

The question of which of the two (equivalent) definitions is the most useful for expressing the semantics of while commands is a polemical one ([18], [24]). However both approaches are complementary and have their equivalent in mechanical sciences which express their laws in two equivalent ways : a dynamic law expressing that a quantity is function of the time (e.g. force F=m.dv/dt) and a static law expressing the conservation of some quantity (e.g. conservation of the momentum m.v).

Remark 2 : forward and backward equations.

As noted in paragraph 2.6.2 backward equations can be used instead of forward ones. A backward system of equations may be obtained by the backward rules of MacCarthy[41]. The semantics of an assignment statement $\{F_1\}\ \overline{X} \leftarrow F(\overline{X})\ \{F_2\}$ is given by $F_1 = F_2 \circ F$. The backward rule for a conditional statement $\{F_1\}$ if $P(\overline{X})$ then $\{F_t\}$... else $\{F_f\}$... fi; is $F_1 = (F_t|P) \sqcup (F_f|\underline{\text{not}}\,P)$. Finally an unconditional jump $\{F_i\}$ go to L; to a constant label L:{F}... leads to F_i=F. Applying these backward rules to the while loop schema :

$\{\varphi\}$ loop: $\{F_1\}$ if $P(\overline{X})$ then $\{F_2\}\ \overline{X} \leftarrow F(\overline{X})\ \{F_3\}$; go to loop; fi; $\{F_4\}$

we get :

$$\begin{cases} \varphi = F_1 \\ F_3 = F_1 \\ F_1 = (F_2|P) \sqcup (F_4|\underline{\text{not}}\,P) \\ F_2 = F_3 \circ F \\ F_4 = \iota \end{cases}$$

which simplifies in : $\varphi = (\varphi \circ F | P) \sqcup (\iota | \underline{\text{not}}\,P)$.

This functional $\varphi(X)$ = if $P(\overline{X})$ then $\varphi(F(\overline{X}))$ fi is often used for the semantics of the while loop (e.g. [44]). The solution to this backward functional $\varphi^{\infty} = \coprod_{k=0}^{\infty} ((F^k) | (\underset{i=0}{\overset{k-1}{\text{AND}}} P \circ (F)^i)\ \underline{\text{and}}\ (\underline{\text{not}}\ P \circ (F)^k))$ is the same as the one of the forward equations.

Preference for forward or backward equations is generally a matter of taste, [18].

6. APPLICATION TO LOGICAL ANALYSIS OF PROGRAMS

Logical analysis of programs ([19],[27]) consists in establishing the logical invariant assertions corresponding to the program which next can be used to verify the program with respect to a specification or to prove that the program contains an error.

6.1 LOGICAL PREDICATES

In this interpretation abstract properties will be logical first order predicates $P(X,\overline{X})$ over the set X of program variables and the set $\overline{X}$ of initial values of these program variables. X and $\overline{X}$ are the free variables in the predicate P.

The assertion $P_i(X,\overline{X})$ associated with a point i of the program can be thought of as describing the values X which the program variables will take at program point i during an execution starting with an initial state $\overline{X}$ of the program variables.

The set of predicates $P(X,\overline{X})$ form a complete lattice $(\sqsubseteq, \bot, \top, \sqcup, \sqcap, \coprod, \prod)$ by choosing respectively (=>, false, true, or, and, OR, AND).

6.2 SYSTEM OF LOGICAL FORWARD EQUATIONS

We use the denotation $\{P(X,\overline{X})\}\ S\ \{Q(X,\overline{X})\}$ to mean that for every X, $\overline{X}$, if $P(X,\overline{X})$ holds prior to execution of the statement S then $Q(X,\overline{X})$ is the strongest post-condition such that the statement S faultless executes and properly terminates leaving the program variables in a final state satisfying Q.

According to the deductive semantics of programming languages ([18],[19],[27]) the following rules serve to associate a system of equations with any sequential program :

Program entry point : $\{(x_i=v_i), i=1..m\}$. The respective initial values of the variables $x_1, \ldots, x_m$ are the symbols $v_1, \ldots, v_m$. The v_i may eventually be Ω denoting the uninitialized value.

Assignment statements : $\{P\}$ i:=E $\{\exists i' \mid P(i \leftarrow i')$ and $(i=E(i \leftarrow i'))\}$; we denote by $\alpha(x \leftarrow y)$ the copy of α in which each occurrence of the variable x is replaced by the variable y. The above rule must be enriched if one wants to take account of the fact that the evaluation of E may fail.

Test statements : {P} if Q then {P and Q}... else {P and (not Q)} ... fi;

Go to statements and labels : L:{ $\underset{i \in pred(L)}{OR}$ P_i}...

Note : Suppose that the meaning of each statement S is given by an *operational semantics* : a state transition function T_S defines for each input state $(X,\overline{X})$ the output state $T_S(X,\overline{X})$ resulting from execution of the statement S. The *deductive semantics* constitutes a structural approximation of the operational semantics, the concretization function Γ being $\Gamma(P) = \{(X,\overline{X}) \mid P(X,\overline{X})\}$. According to hypothesis 4.3.0.4.b the rule $\{P_I\}\ S\ \{P_O\}$ that is $P_O=D_S(P_I)$ must be validated by showing that for any P_I we have $T_S(\Gamma(P_I)) \subseteq \Gamma(D_S(P_I))$. However this validation is too liberal since it does not require the deductive semantics to be equivalent to the operational semantics. More precisely in order for termination of programs to be provable with the formalism of the deductive semantics, one must also show that the operational semantics is a structural approximation of the deductive semantics. The concretization of a set of states σ is the characteristic predicate $\Gamma^{-1}(\sigma)$ of this set, that is $\{x \in \sigma\} \Longleftrightarrow \Gamma^{-1}(\sigma)(x)$. The state transition function T_S is validated with respect to the rule $\{P_I\}\ S\ \{P_O\}$ that is $P_O=D_S(P_I)$ by showing that for any set of states σ we have $D_S(\Gamma^{-1}(\sigma)) \Longrightarrow \Gamma^{-1}(T_S(\sigma))$. Finally equivalence of these semantics implies $\{\forall P_I, T_S(\Gamma(P_I))=\Gamma(D_S(P_I))\}$ or $\{\forall \sigma, D_S(\Gamma^{-1}(\sigma)) \Longleftrightarrow \Gamma^{-1}(T_S(\sigma))\}$. *End of Note.*

We will illustrate the above rules for the very simple program (over the integers $\mathbb{I}^* = \mathbb{I} \cup \{\Omega\}$) :

```
{P1}
{P2} while x≥y do
          x:=x-y;
{P3}
{P4} od;
```

fig.6.2.a

Rewritting this program segment with branch and test primitives, and applying the logical forward rules we get a system of equations which can be simplified as follows :

$$\begin{cases} P_1 = (x=x_0) \text{ and } (y=y_0) \\ P_2 = (P_1 \text{ or } P_3) \text{ and } (x\geq y) \\ P_3 = \{\exists x'\in\mathbb{I}^* \mid P_2(x') \text{ and } (x=x'-y)\} \\ P_4 = (P_1 \text{ or } P_3) \text{ and } (x<y) \end{cases}$$

fig.6.2.b

6.3 OPTIMAL INVARIANTS

According to theorem 2.3.6 the system of equations fig.6.2.b of the form $P=F(P)$ has a least solution P^{opt} (least for ordering $\sqsubseteq_4$ that is $\Rightarrow_4$). Since P^{opt} is the infimum of the complete lattice of fixed points of F we have $\{\forall P | P=F(P)\}$, $P^{opt} \Rightarrow P$. Therefore we call P^{opt} the set of *optimal invariants* since they imply any other solution to the system of equations.

The optimal invariants P^{opt} are the limit of any chaotic transform of the ascending approximation sequence starting from the infimum false :

Initialization :

$$\left[P_i^0 = \text{false}, \; i=1..4 \right.$$

Step 1 :

$$\left[\begin{array}{l} P_1^1 = (x=x_0) \text{ and } (y=y_0) \\ P_2^1 = (P_1^1 \text{ or } P_3^0) \text{ and } (x\geq y) = (P_1^1 \text{ or false}) \text{ and } (x\geq y) = (x=x_0) \text{ and } (y=y_0) \text{ and } (x_0\geq y_0) \\ P_3^1 = \{\exists x' | P_2^1(x') \text{ and } (x=x'-y)\} = \{\exists x' | (x_0\geq y_0) \text{ and } (x'=x_0) \text{ and } (y=y_0) \text{ and } (x=x'-y)\} \\ \quad\;\; = (x_0\geq y_0) \text{ and } (x=x_0-y_0) \text{ and } (y=y_0) \\ P_4^1 = (P_1^1 \text{ or } P_3^0) \text{ and } (x<y) = (x=x_0) \text{ and } (y=y_0) \text{ and } (x_0<y_0) \end{array}\right.$$

Step 2 :

$$\left[\begin{array}{l} P_1^2 = (x=x_0) \text{ and } (y=y_0) \\ P_2^2 = ((x_0\geq y_0) \text{ and } (x=x_0) \text{ and } (y=y_0)) \text{ or } ((x_0\geq y_0) \text{ and } (x_0-y_0\geq y_0) \text{ and } (x=x_0-y_0) \text{ and } (y=y_0)) \\ P_3^2 = ((x_0\geq y_0) \text{ and } (x=x_0-y_0) \text{ and } (y=y_0)) \text{ or } ((x_0\geq y_0) \text{ and } (x_0-y_0\geq y_0) \text{ and } (x=x_0-2y_0) \text{ and } (y=y_0)) \\ P_4^2 = ((x_0<y_0) \text{ and } (x=x_0) \text{ and } (y=y_0)) \text{ or } ((x_0\geq y_0) \text{ and } (x_0-y_0<y_0) \text{ and } (x=x_0-y_0) \text{ and } (y=y_0)) \end{array}\right.$$

By computing the first terms of the approximation sequence we seek to discover the general term P^i of the approximation sequence. We have found :

$$\left[\begin{array}{l} P_1^i = (x=x_0) \text{ and } (y=y_0) \\ P_2^i = \underset{j=0}{\overset{i-1}{\text{OR}}} (\underset{k=0}{\overset{j}{\text{AND}}}(x_0-ky_0\geq y_0) \text{ and } (x=x_0-jy_0) \text{ and } (y=y_0)) \\ P_3^i = \underset{j=0}{\overset{i-1}{\text{OR}}} (\underset{k=0}{\overset{j}{\text{AND}}}(x_0-ky_0\geq y_0) \text{ and } (x=x_0-(j+1)y_0) \text{ and } (y=y_0)) \\ P_4^i = \underset{j=0}{\overset{i-1}{\text{OR}}} (\underset{k=0}{\overset{j-1}{\text{AND}}}(x_0-ky_0\geq y_0) \text{ and } (x_0-jy_0<y_0) \text{ and } (x=x_0-jy_0) \text{ and } (y=y_0)) \end{array}\right.$$

It is easy to verify that P^1 and P^2 are of the above form. According to 4.4.1, supposing P^i to be correct and substituting in the equations we must show that we obtain P^{i+1} :

$$\left[\begin{array}{l} P_1^{i+1} = (x=x_0) \text{ and } (y=y_0) \\ P_2^{i+1} = (P_1^{i+1} \text{ and } x\geq y) \text{ or } (P_3^i \text{ and } x\geq y) \\ \quad\;\; = \{(x_0\geq y_0) \text{ and } (x=x_0) \text{ and } (y=y_0)\} \\ \qquad \text{or} \\ \qquad \{\underset{j=0}{\overset{i-1}{\text{OR}}} (\underset{k=0}{\overset{j}{\text{AND}}}(x_0-ky_0\geq y_0) \text{ and } (x=x_0-(j+1)y_0) \text{ and } (y=y_0)) \text{ and } (x\geq y)\} \\ \quad\;\; = \{(x_0\geq y_0) \text{ and } (x=x_0) \text{ and } (y=y_0)\} \\ \qquad \text{or} \\ \qquad \{\underset{j'=1}{\overset{i}{\text{OR}}}(\underset{k=0}{\overset{j'}{\text{AND}}}(x_0-ky_0\geq y_0) \text{ and } (x=x_0-j'y_0) \text{ and } (y=y_0))\} \end{array}\right.$$

$$= \underset{j=0}{\overset{i}{\underline{OR}}} \left(\underset{k=0}{\overset{j}{\underline{AND}}} (x_0 - ky_0 \geq y_0) \ \underline{and} \ (x = x_0 - jy_0) \ \underline{and} \ (y = y_0) \right)$$

$$P_3^{i+1} = \{ \exists x' \mid P_2^{i+1}(x') \ \underline{and} \ (x = x' - y) \}$$

$$= \underset{j=0}{\overset{i}{\underline{OR}}} \{ \exists x' \mid \underset{k=0}{\overset{j}{\underline{AND}}} (x_0 - ky_0 \geq y_0) \ \underline{and} \ (x' = x_0 - jy_0) \ \underline{and} \ (y = y_0) \ \underline{and} \ (x = x' - y) \}$$

$$= \underset{j=0}{\overset{i}{\underline{OR}}} \left(\underset{k=0}{\overset{j}{\underline{AND}}} (x_0 - ky_0 \geq y_0) \ \underline{and} \ (x = x_0 - (j+1)y_0) \ \underline{and} \ (y = y_0) \right)$$

$$P_4^{i+1} = (P_1^{i+1} \ \underline{and} \ (x<y)) \ \underline{or} \ (P_3^{i} \ \underline{and} \ (x<y))$$

$$= \{ (x = x_0) \ \underline{and} \ (y = y_0) \ \underline{and} \ (x_0 < y_0) \}$$

$$\underline{or}$$

$$\{ \underset{j=0}{\overset{i-1}{\underline{OR}}} \left(\underset{k=0}{\overset{j}{\underline{AND}}} (x_0 - ky_0 \geq y_0) \ \underline{and} \ (x = x_0 - (j+1)y_0) \ \underline{and} \ (y = y_0) \ \underline{and} \ (x<y) \right\}$$

$$= \underset{j=0}{\overset{i}{\underline{OR}}} \left(\underset{k=0}{\overset{j-1}{\underline{AND}}} (x_0 - ky_0 \geq y_0) \ \underline{and} \ (x_0 - jy_0 < y_0) \ \underline{and} \ (x = x_0 - jy_0) \ \underline{and} \ (y = y_0) \right)$$

Since now we have proved the general form P(i) of the ith term of the approximation sequence to be correct, the optimal invariants are obtained by : $P^{opt} = \lim_{i \to \infty} P^i$ which directly results in the optimal invariants :

$$P_1^{opt} = (x = x_0) \ \underline{and} \ (y = y_0)$$

$$P_2^{opt} = \underset{j=0}{\overset{\infty}{\underline{OR}}} \left(\underset{k=0}{\overset{j}{\underline{AND}}} (x_0 - ky_0 \geq y_0) \ \underline{and} \ (x = x_0 - jy_0) \ \underline{and} \ (y = y_0) \right)$$

$$P_3^{opt} = \underset{j=0}{\overset{\infty}{\underline{OR}}} \left(\underset{k=0}{\overset{j}{\underline{AND}}} (x_0 - ky_0 \geq y_0) \ \underline{and} \ (x = x_0 - (j+1)y_0) \ \underline{and} \ (y = y_0) \right)$$

$$P_4^{opt} = \underset{j=0}{\overset{\infty}{\underline{OR}}} \left(\underset{k=0}{\overset{j-1}{\underline{AND}}} (x_0 - ky_0 \geq y_0) \ \underline{and} \ (x_0 - jy_0 < y_0) \ \underline{and} (x = x_0 - jy_0) \ \underline{and} \ (y = y_0) \right)$$

fig.6.3.a

Note : It is remarkable that the approximation sequence really describes the execution of the program. P^i describes the state of variables after k cycles in the loop for any k less than or equal to i. Yet in general the approximation sequence is a means of computation which differs from usual execution since all possible program paths are followed in parallel and the paths are initiated from all program points (and not only from program entry points). Later on we will show that this corresponds to symbolic execution. *End of Note.*

6.4 PROOF OF TOTAL CORRECTNESS

The system of equations is generated directly from the program text according to the rules of the deductive semantics of the language. Therefore the optimal invariants are independant of any user provided input/output specification and reflect what is actually happening during the computation, as opposed to what is supposed to be happening.

Suppose now that the intended behaviour of the program π is specified by means of an input specification $\Phi(X)$ and an output specification $\psi(X,\overline{X})$. The intention is that for any initial values $\overline{X}$ of the program variables satisfying the input specification $\Phi(\overline{X})$, the program terminates with final values $\overline{Y}$ of the variables satisfying $\psi(\overline{Y},\overline{X})$.

The verification of correctness of a program π for input/output specifications Φ and ψ then consists in constructing the system of assertion equations P=F(P) of π, and finding its optimal solution P^{opt}, and next in proving that for every input $\overline{X}$ such that $\Phi(\overline{X})$ is true, there exists a haltpoint h, there exist output values $\overline{Y}$ of the variables such that :

$P_h^{opt}(\overline{Y},\overline{X})$ and $(P_h^{opt}(\overline{Y},\overline{X}) \Longrightarrow \psi(\overline{Y},\overline{X}))$. This requires that the program terminates at some halt-point h with a state $\overline{Y}$ of the program variables satisfying the output specification ψ. In formulas (adapted from [32]) we must prove :

$$\{(\forall\overline{X}|\Phi(\overline{X})), \exists h, \exists\overline{Y} \mid P_h^{opt}(\overline{Y},\overline{X}) \text{ and } \psi(\overline{Y},\overline{X})\}$$

(Note that it is absolutely necessary that $P^{opt}=lfp(F)$ since the optimal invariant P_i^{opt} is the only invariant which describes the exact domain of the values $\overline{Y}$ of the variables which occur at i during execution of the program with input $\overline{X}$).

Example : suppose that one wants to prove that for any input value (x_0,y_0) satisfying the input specification $\Phi(x_0,y_0) = \{(x_0\geq 0) \text{ and } (y_0\geq 0)\}$ the program of fig.6.2.a terminates with output specification $\psi(x,y,x_0,y_0) = \{(x\geq 0) \text{ and } (y\geq 0) \text{ and } (x<y)\}$. Given the set of optimal invariants of fig.6.3.a we must prove :

$$\{(\forall x_0|x_0\geq 0),(\forall y_0|y_0\geq 0),\exists(x,y) \mid P_4^{opt}(x,y,x_0,y_0) \text{ and } \psi(x,y,x_0,y_0)\}$$

After trivial simplifications this consists in proving that $(x_0\geq 0)$ and $(y_0\geq 0)$ implies :

$$\{\exists j\geq 0 | \underset{k=0}{\overset{j-1}{\text{AND}}}(x_0-ky_0\geq y_0) \text{ and } (x_0-jy_0<y_0) \text{ and } (x_0\geq 0) \text{ and } (y_0\geq 0)\}$$

We know from arithmetic that $\forall x_0\geq 0, \forall y_0>0$, $\exists q$, $\exists r$ such that $x_0-qy_0=r$ and $0\leq r<y_0$. Choosing j=q in the above formula it remains to prove : $\underset{k=0}{\overset{q-1}{\text{AND}}}(x_0-ky_0\geq y_0)$.

But $x_0\geq qy_0$ then for any k satisfying $q>k\geq 0$ we have $q\geq(k+1)$ thus $qy_0\geq(k+1)y_0$ and by transitivity $x_0\geq(k+1)y_0$ we complete the proof of termination and correctness when $(x_0\geq 0)$, $(y_0>0)$.

However in the remaining case $x_0\geq 0$, $y_0=0$ the program is obviously incorrect since we cannot have $((x_0<0) \text{ and } (x_0\geq 0))$. For the same reason $P_4^{opt}(x,y,x_0,0)$ is always false when $(x_0\geq 0)$ hence the program must enter an infinite loop for such input values. *End of Example.*

6.5 APPROXIMATE INVARIANTS, SYSTEMS OF INEQUALITIES AND PROOFS OF PARTIAL CORRECTNESS

Most program proving methods use inequalities of the form : $P_i \Leftarrow f_i(P_1,\ldots,P_n)$, i=1..n whereas we used the equations : $P_i=f_i(P_1,\ldots,P_n)$, i=1..n.

For example, instead of {i>0} i:=i+2 {i>2} one can legally write a less precise assertion such as {i>0} i:=i+2 {i>1} since the strongest post-condition resulting from the pre-condition {i>0} is {i>2} which implies {i>1}.

This reasoning is justified by theorem 4.3.0.2 since any set of *approximate invariants* P solution to the system of inequalities $P \Leftarrow F(P)$ correctly approximates the set P^{opt} of optimal invariants that is $P^{opt} \Longrightarrow P$. More precisely a program π is partially correct with respect to $\Phi(\overline{X})$ and $\psi(X,\overline{X})$ if for every input $\overline{X}$ such that $\Phi(\overline{X})$ is true, whenever the program terminates at some haltpoint h with some final value $\overline{Y}$ of the variables X (that is $\{\exists h, \exists\overline{Y}|P_h^{opt}(\overline{Y},\overline{X})\}$) the output specification must be true (that is $\{P_h^{opt}(\overline{Y},\overline{X}) \Rightarrow \psi(\overline{Y},\overline{X})\}$). In order to guarantee that for these particular h and $\overline{Y}$ the optimal $P_h^{opt}(\overline{Y},\overline{X})$ implies $\psi(\overline{Y},\overline{X})$ it suffices to find some set P of approximate invariants such that $\{\forall h,\forall\overline{Y},P_h(\overline{Y},\overline{X}) \Rightarrow \psi(\overline{Y},\overline{X})\}$ since we know that $\{\forall h,\forall\overline{Y},P_h^{opt}(\overline{Y},\overline{X}) \Rightarrow P_h(\overline{Y},\overline{X})\}$.

Finally, the program π is partially correct with respect to Φ and ψ if and only if :

$$\{(\exists P|P \Leftarrow F(P)), (\forall\overline{X}|\Phi(\overline{X})), \forall h, \forall\overline{Y}, (P_h(\overline{Y},\overline{X}) \Longrightarrow \psi(\overline{Y},\overline{X}))\}$$

which is the condition given in [32].

Example : Suppose we want to prove the partial correctness of the program of fig.6.2.a with respect to the input specification $\Phi(x_0,y_0) = \{(x_0\geq 0) \text{ and } (y_0\geq 0)\}$ and the output specification $\psi(x,y,x_0,y_0) = \{y>x\geq 0\}$.

Choosing the loop invariant P_2 to be : $P_2=(x\geq y)$ <u>and</u> $(y=y_0)$ and propagating in the equations we get :

$P_3 = (x\geq 0)$ <u>and</u> $(y=y_0)$

$P_4 = (x<y)$ <u>and</u> $(y=y_0)$ <u>and</u> $[(x=x_0)$ <u>or</u> $(x\geq 0)]$

It is easy to verify that $\{(P_1$ <u>or</u> $P_3)$ <u>and</u> $(x\geq y)\} \Rightarrow P_2$, so that P_2 is a correct approximate loop invariant. Finally we have $\{V(x_0,y_0),\Phi(x_0,y_0)$ <u>and</u> $P_4(x,y,x_0,y_0)\} \Rightarrow \psi(x,y,x_0,y_0)$ so that the program is proved to be partially correct (although it does not terminate for y=0).

The verification of program partial correctness has been shown to be amenable to mechanization (see a.o. [37]). Since the automatic discovery of optimal invariants is an unsolvable problem the programmer must provide inductive approximate invariants which cut the loops in the program, that is provide the invariants which recursively depend on themselves in the equations. This in fact provides the entire solution to the inequalities since a simple propagation in the equations permits the remaining invariants to be deduced.

Thus proving partial correctness of programs consists in verifying that an approximate solution to the system of logical equations corresponding to the program is correct. Next the solution is used to prove the output specification when assuming the input specification.
End of Example.

Note : Let π be a program for which the corresponding system of equations is P=F(P). The termination condition with respect to an input specification Φ was given at paragraph 6.4 as $\{(\forall\bar{X}|\Phi(\bar{X})),\exists h,\exists\bar{Y}|P_h^{opt}(\bar{Y},\bar{X})\}$. But according to theorem 2.3.6 $P^{opt}=\underline{AND}\{P|F(P)\Longrightarrow P\}$ and the termination condition becomes :

$$\{(\forall\bar{X}|\Phi(\bar{X}),\ \exists h,\ \exists\bar{Y}\ |\ \underline{AND}\{P_h(\bar{Y},\bar{X})|F(P)\Longrightarrow P\}\}$$
$$=\ \{(\forall P|P\Longleftarrow F(P)),\ (\forall\bar{X}|\Phi(\bar{X})),\ \exists h,\ \exists\bar{Y}\ |\ P_h(\bar{Y},\bar{X})\}$$

which is the termination condition of Katz and Manna [32].

However they observed that this last condition is not utilizable in practice since it is expressed in terms of every possible set of approximate invariants satisfying the system of inequalities $F\Longleftarrow F(P)$ which admits infinitely many solutions. This fact is not surprising at all since this condition is based on theorem 2.3.6 which is not constructive. *End of Note.*

6.6 SYMBOLIC EXECUTION

Symbolic execution is a widely used program analysis technique (e.g. [7], [8], [21], [22], [31], [38], [55], [57], [67]). In light of the fixed point approach to analysis of programs we show that symbolic execution may be used to compute the set of optimal invariants.

6.6.1 SYMBOLIC CONTEXTS

The invariant associated with a program point i can be expressed in the normal form $P_i = \underset{j\in\Delta}{\underline{OR}}\, p_j$ where each p_j is of the form $(Q_j$ and $(x_1=E_{1j})$ <u>and</u>...<u>and</u> $(x_m=E_{mj}))$. Each p_j describes a program path which may lead to the program point i. For each program path p_j an assertion Q_j states the condition which had to be satisfied in order for that path to be executed. At point i on that path the value of the program variable x_k (k=1..m) is given by E_{kj} . E_{kj} is a formal expression depending on the formal symbols $v_1,\dots,v_m$ which represent the arbitrary initial values of the variables $x_1,\dots,x_m$ on program entry. No x_k can appear as a free variable neither in Q_j nor in the E_{kj}. Changing the notations we will call P_i a symbolic context and rewrite it in the shape $P_i=\{p_j|j\in\Delta\}$ with $p_j=\langle Q_j,E_{1j},\dots,E_{1m}\rangle$.

6.6.2 SYSTEM OF SYMBOLIC FORWARD EQUATIONS

Let $P=\{<Q_j,E_{1j},\dots,E_{mj}>|j\in\Delta\}$ be a symbolic context, $E(x_1,\dots,x_m)$ be an expression depending on the program variables $x_1,\dots,x_m$ and $B(x_1,\dots x_m)$ be a boolean expression. We denote by $P(x_k\leftarrow E(x_1,\dots,x_m))$ the assignment of the symbolic value of E in context P to the variable x_k that is $\{<Q_j,E_{1j},\dots,E(E_{1j},\dots,E_{mj}),\dots,E_{mj}>|j\in\Delta\}$. The same way P $\underline{\text{and}}$ $B(x_1,\dots,x_m)$ will denote the context $\{<Q_j\ \underline{\text{and}}\ B(E_{1j},\dots,E_{mj}),E_{1j},\dots,E_{mj}>|j\in\Delta\}$.

The system of equations corresponding to the program segment :

```
       {P0}
loop : {P1}   if x≥y then
       {P2}      x:=x-y;
       {P3}      go to loop;
              fi;
       {P4}
```

fig.6.6.2.a

would be :

$$\begin{cases} P_0 = \{<\underline{true},x_0,y_0>\} \\ P_1 = P_0 \sqcup P_3 \\ P_2 = P_1\ \underline{\text{and}}\ (x\geq y) \\ P_3 = P_2(x\leftarrow(x-y)) \\ P_4 = P_1\ \underline{\text{and}}\ (x<y) \end{cases}$$

fig.6.6.2.b

The equations can be justified with respect to the rules of paragraph 6.2. For example, the output predicate P_0 corresponding to the input predicate $P_I=\underset{j\in\Delta}{\text{OR}}(Q_j\ \underline{\text{and}}\ (x_1=E_{1j})\ \underline{\text{and}}\dots\underline{\text{and}}\ (x_m=E_{1m}))$ after the assignment statement $x_k := E(x_1,\dots,x_m)$ is $P_0=\{\exists x'|P_I(x_k\leftarrow x')\ \underline{\text{and}}\ (x_k=E(x_1,\dots,x',\dots x_m))\}$. Since x_k is not a free variable in Q_j or any E_{kj}, we have :

$P_0=\underset{j\in\Delta}{\text{OR}}(Q_j\ \underline{\text{and}}(x_1=E_{1j})\ \underline{\text{and}}\dots\underline{\text{and}}\ (x'=E_{kj})\ \underline{\text{and}}\dots\underline{\text{and}}\ (x_m=E_{mj})\ \underline{\text{and}}\ (x_k=E(x_1,\dots,x',\dots,x_m)))$.

Eliminating the free variables in E we get :

$P_0=\underset{j\in\Delta}{\text{OR}}(Q_j\ \underline{\text{and}}\ (x_1=E_{1j})\ \underline{\text{and}}\dots\underline{\text{and}}\ (x_k=E(E_{1j},\dots,E_{kj},\dots,E_{mj}))\ \underline{\text{and}}\dots\underline{\text{and}}\ (x_m=E_{mj}))$.

Therefore using the denotations of symbolic contexts we have the rule for assignment : $\{P\}\ x_k:=E(x_1,\dots,x_m)\ \{P(x_k\leftarrow E(x_1,\dots,x_m))\}$. The other rules can be justified in the same way. In particular the operation $\sqcup$ describes the union $P\sqcup Q$ of two symbolic contexts $P=\{p_1,\dots,p_r\}$ and $Q=\{q_1,\dots,q_s\}$ that is the set $\{p_1,\dots,p_r,q_1,\dots,q_s\}$ where superfluous equivalent program paths are eliminated whereas inaccessible paths (the path condition of which is false) are removed.

6.6.3 SYMBOLIC EXECUTION TREE

Computing the Gauss-Seidel's transform of the ascending approximation sequence, we get as in paragraph 6.3 :

Initialization :

$$\left[P_0^0=P_1^0=P_2^0=P_3^0=P_4^0 = \emptyset \right.$$

Step1 :

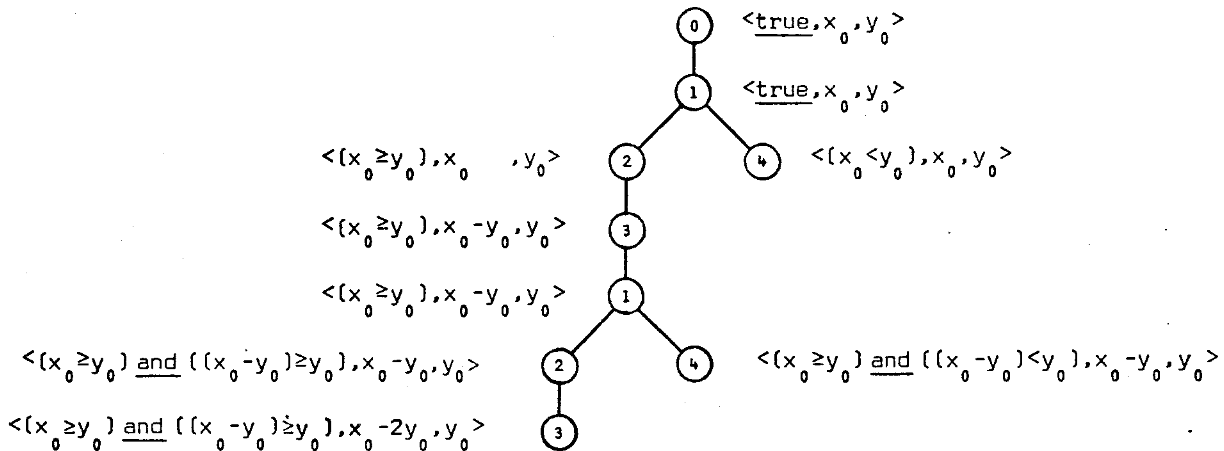
$$\begin{cases} P^1_0 = \{<\underline{true},x_0,y_0>\} \\ P^1_1 = P^1_0 \sqcup P^0_3 = \{<\underline{true},x_0,y_0>\} \\ P^1_2 = P^1_1 \ \underline{and}\ (x\geq y) = \{<(x_0\geq y_0),x_0,y_0>\} \\ P^1_3 = P^1_2(x\leftarrow x-y) = \{<(x_0\geq y_0),x_0-y_0,y_0>\} \\ P^1_4 = P^1_1 \ \underline{and}\ (x<y) = \{<(x_0<y_0),x_0,y_0>\} \end{cases}$$

fig.6.6.3.a

Step2 :

$$\begin{cases} P^2_0 = \{<\underline{true},x_0,y_0>\} \\ P^2_1 = \{<\underline{true},x_0,y_0>,\ <(x_0\geq y_0),x_0-y_0,y_0)>\} \\ P^2_2 = \{<(x_0\geq y_0),x_0,y_0>,\ <(x_0\geq y_0)\ \underline{and}\ ((x_0-y_0)\geq y_0),x_0-y_0,y_0>\} \\ P^2_3 = \{<(x_0\geq y_0),x_0-y_0,y_0>,\ <(x_0\geq y_0)\ \underline{and}\ ((x_0-y_0)\geq y_0),x_0-2y_0,y_0>\} \\ P^2_4 = \{<(x_0<y_0),x_0,y_0>,\ <(x_0\geq y_0)\ \underline{and}\ ((x_0-y_0)<y_0),x_0-y_0,y_0>\} \end{cases}$$

so that at iteration 2 we have built the following symbolic execution tree ([22]) :



We have represented the symbolic context P_i associated with program point i by the set of paths associated with each of the nodes labelled i in the above execution tree. Equivalently we could have represented the symbolic context associated with program point i by the maximal subtree (of the above symbolic tree) the leaves of which are labelled i. Then the union $\sqcup$ of symbolic contexts performed at junction program points would be the merging of symbolic execution trees.

It is clear that the computation of the next terms in the approximation sequence would cause the symbolic execution tree to grow. Without particular hypothesis on x_0 and y_0 this process would converge to the optimal invariants in infinitely many steps. The various mathematical techniques for dealing with infinite approximation sequences reviewed at paragraph 4 can be used to cope with infinite execution trees. The verification of the correctness of the optimal invariants (4.4.1) and the verification of the correctness of approximate invariants have been illustrated at paragraphs 6.3 and 6.5 respectively. We now illustrate the technique 4.2 and next 4.1.2.

6.6.4 VERIFICATION OF PROPERTIES OF OPTIMAL SYMBOLIC CONTEXTS

Let us prove the trivial fact that assertion $\{(\forall k|1\le k\le(x_0-x)/y)(x_0\ge ky)\}$ holds at point {3} of the program given at fig.6.6.2.a. According to Scott's induction (4.2) the basis $P(F(\perp))$ and the induction step $\{(\forall X)(P(X))\}\Rightarrow\{P(F(X))\}$ imply $P(lfp(F))$. Therefore the proof is the following :

Basis : After step 1 P_3^1 is equal to $\{<(x_0\ge y_0),x_0-y_0,y_0>\}$ so that trivially $(\forall k|1\le k\le 1)$ we have $x_0\ge ky$.

Induction step : We assume that at step ℓ the symbolic context P_3^ℓ is equal to $\{<p_i,x_i,y_i>|i\in D\}$ with the induction hypothesis that for any i of D we have $\{(\forall k|1\le k\le(x_0-x_i)/y_i),(x_0\ge ky_i)\}$.

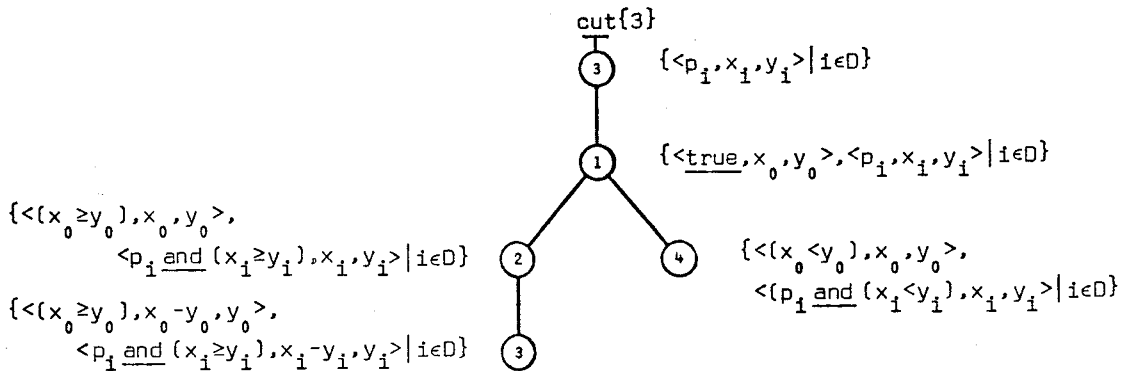
The equations of fig.6.6.2.b allow the computation of $P_3^{\ell+1}$:

$$P_3^\ell = \{<p_i,x_i,y_i>|i\in D\}$$
$$P_1^{\ell+1} = P_0 \sqcup P_3^\ell = \{<\underline{true},x_0,y_0>,<p_i,x_i,y_i>|i\in D\}$$
$$P_2^{\ell+1} = P_1^{\ell+1}\ \underline{and}(x\ge y) = \{<(x_0\ge y_0),x_0,y_0>,<(p_i\ \underline{and}\ (x_i\ge y_i)),x_i,y_i>|i\in D\}$$
$$P_3^{\ell+1} = P_2^{\ell+1}(x\leftarrow x-y) = \{<(x_0\ge y_0),x_0-y_0,y_0>,<(p_i\ \underline{and}\ (x_i\ge y_i)),x_i-y_i,y_i>|i\in D\}$$

We must now prove that the hypothesis holds for all paths of $P_3^{\ell+1}$. This is trivial for the path $<(x_0\ge y_0),x_0-y_0,y_0>$. Otherwise we have to prove $\{(\forall k|1\le k\le(x_0-x_i)/y_i+1),(x_0\ge ky_i)\}$.

According to the induction hypothesis, this condition is true for $1\le k\le(x_0-x_i)/y_i$. Finally for $k=(x_0-x_i)/y_i+1$ the path condition $x_i\ge y_i$ implies $x_0\ge ky_i$.

This approach is implicitly used in the technique of "cut-trees" of Hantler and King [22]. Indeed the induction step can be understood as consisting in reasoning on the cut tree for {3}.



6.6.5 DISCOVERY OF OPTIMAL SYMBOLIC CONTEXTS

The technique of the difference equations (4.1.2) permits the optimal invariants to be discovered ; the least fixed point of X=F(X) is obtained by solving the difference equations $S^1=F(\perp)$ and $S^{\ell+1}=F(S^\ell)$, that is by expressing S^ℓ as a function $f(\ell)$ of ℓ. Then $lfp(F)=\lim_{\ell\to\infty} f(\ell)$.

The first iteration of the Gauss-Seidel's transform of the ascending approximation sequence (fig.6.6.3.a) leads to the basis for the difference equations : $P_1^1=\{<\underline{true},x_0,y_0>\}$. Then we establish a recurrence relationship between $P_1^{\ell+1}$ and P_1^ℓ using the equations fig.6.6.2.b :

$$P_1^\ell = \{<p_{i,\ell},x_{i,\ell},y_{i,\ell}>|i\in D(\ell)\}$$
$$P_2^\ell = \{<p_{i,\ell}\ \underline{and}\ (x_{i,\ell}\le y_{i,\ell}),x_{i,\ell},y_{i,\ell}>|i\in D(\ell)\}$$

$$P_3^{\ell} = \{<p_{i,\ell}\ \underline{\text{and}}\ (x_{i,\ell} \le y_{i,\ell}), x_{i,\ell}-y_{i,\ell}, y_{i,\ell}> \mid i \in D(\ell)\} = \bar{\Delta}(P_1^{\ell})$$

where $\bar{\Delta}\{<p_{i,\ell}, x_{i,\ell}, y_{i,\ell}> \mid i \in D(\ell)\} = \{\Delta(<p_{i,\ell}, x_{i,\ell}, y_{i,\ell}>) \mid \ell \in D(\ell)\}$ and

$$\Delta(<p_{i,\ell}, x_{i,\ell}, y_{i,\ell}>) = <p_{i,\ell}\ \underline{\text{and}}\ (x_{i,\ell} \le y_{i,\ell}), x_{i,\ell}-y_{i,\ell}, y_{i,\ell}>.$$

$P_1^{\ell+1} = P_0 \sqcup \bar{\Delta}(P_1^{\ell})$ since P_0 is constant and $P_3^{\ell} = \bar{\Delta}(P_1^{\ell})$.

Hence the recurrence relations defining P_1 are :

$$\begin{cases} P_1^1 = P_0 \\ P_1^{\ell+1} = P_0 \sqcup \bar{\Delta}(P_1^{\ell}) \end{cases}$$

Using the property that $\bar{\Delta}$ is distributive over $\sqcup$, the solution to these regular equations is :

$$P_1^{\ell} = \coprod_{i=0}^{\ell-1} \bar{\Delta}^i(P_0) = \{\Delta^i(P_0) \mid i=0..(\ell-1)\}$$

It remains now to determine the multivalued function Δ^i. This is done using the fact that Δ^i is defined recursively by $\Delta^{i+1} = \Delta \circ \Delta^i$ and Δ^0 is the identity function. We have : $\Delta^0(P_0) = <\underline{\text{true}}, x_0, y_0> = <p_0, x_0, y_0>$. Let $\Delta^i(P_0) = <p_i, x_i, y_i>$ then $\Delta^{i+1}(P_0) = <p_i\ \underline{\text{and}}\ (x_i \le y_i), x_i-y_i, y_i> = <p_{i+1}, x_{i+1}, y_{i+1}>$.

These recurrence relations may be solved directly, yielding $(y_i = y_0)$, $(x_i = x_0 - iy_0)$ and since $p_0 = \underline{\text{true}}$ and $p_{i+1} = p_i\ \underline{\text{and}}\ ((x_0 - iy_0) \le y_0)$ we have $p_i = \underline{\text{AND}}_{j=0}^{i-1}((x_0 - jy_0) \le y_0)$ and therefore $P_1^{\ell} = \{\Delta^i(P_0) \mid i=0..\ell-1\} = \{<\underline{\text{AND}}_{j=0}^{i-1}((x_0 - jy_0) \le y_0), x_0 - iy_0, y_0> \mid i=0..\ell-1\}$.

Thus the optimal solution to equations fig.6.6.2.b. is given by $P_1^{opt} = \lim_{\ell \to \infty} P_1^{\ell} = \{<\underline{\text{AND}}_{j=0}^{i-1}((x_0 - jy_0) \le y_0), x_0 - iy_0, y_0> \mid i \ge 0\}$. The other symbolic contexts are obtained by a straightforward propagation of this value in the equations fig.6.6.2.b and we get the optimal invariants of fig.6.3.a.

The above approach is implicitely used in [8], [21], in the "algorithmic approach" of Katz & Manna [32], etc. The main advantage of the use of symbolic execution and difference equations is that they lead to the optimal invariants. However for more complex programs the resolution of the difference equations is often difficult and can be a problem of considerable intellectual depth.

7. APPLICATION TO COMPILE-TIME VERIFICATION OR DISCOVERY OF PROGRAM PROPERTIES

Traditionally compilers do not necessitate intervention of programmers during the compilation of programs so that compile-time analysis of programs must be entirely automatic. However since compilers happen to seek for not decidable properties of programs the approximation techniques 4.3.1 and 4.3.2 play a central role. These approximation techniques have been designed so that (at least but not necessarily at the most) all states which may occur during execution are discovered. For example the fact that an integer variable x can only take the values 0 and 11 may be correctly approximated by $0 \le x \le 255$. However it should be clear that this incomplete analysis is correct and acceptable to compilers which never need full knowledge of the properties of the compiled programs.

7.1 APPLICATION TO GLOBAL DATA FLOW ANALYSIS

Global data flow analysis techniques (see references in [62]) are used for analyzing computer programs for the purpose of code optimization. One such problem is that of determining which variables are *live* (i.e. which variables will be used again) at any given point in the

program ([34]).

Assuming that a program has been represented by its flow graph a path in this graph is said to be *definition clear with respect to* X, or X-*clear*, if there is no assignment to X in any node on that path. A variable X is *live* at point p in the flow graph if there exists an X-clear path from p to a use of X. Thus X is live at p if its value at p may be used before it is redefined.

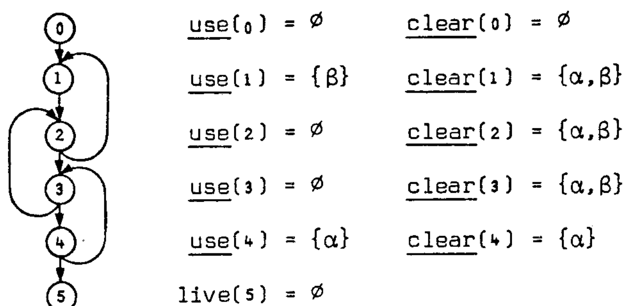
Let live(b) be the set of all variables which are live on entry to node b. The global sets live(b) can be defined in terms of two sets which contain strictly local information. Given a node b in the flow graph use(b) is the set of variables X such that there is an X-clear path from the entry of b to a use of X within b. Let clear(b) be the set of variables X for which the path through the node b is definition-clear with respect to X. Now there is an X-clear path from the entry of b to a use of X if and only if there exists such a path to a use within b or there exists an X-definition clear path through b to a successor of b and there to a use. Hence the set live(b) is defined by the backward equation :

$$\text{live}(b) = \text{use}(b) \cup \bigcup_{x \in \text{succ}(b)} (\text{clear}(b) \cap \text{live}(x)) \qquad \text{fig.7.1.a}$$

For an exit node e which has no successors and contains no command we have live(e)=∅. The least solutions (with respect to set inclusion ⊂) to the systems of equations are preferred. (Consider for example a program such as begin x:=1; while...do...od; y:=x; end where x and y do not appear in the loop. The smallest solution will only consider x to be live in the loop whereas greater solutions might also consider that y is live in the loop).

There are two classical techniques for solving the equations. The Cocke-Allen interval analysis (typified by [3],[9]) is a graphical formulation for an algorithm to formally solve the equations (see 4.1.1). It is applicable only to a limited class of recursive equations (corresponding to "reducible" programs) and to a limited class of lattices. More generally the equations can be solved by iterative methods (typified by Hecht & Ullman[23]) which proceed by successive approximations (see 2.5.3). The utilized lattices are usually finite or satisfy the chain condition (2.4.5) so that convergence is guaranteed.

Let us solve the live-equations for the following irreducible flow graph with two variables α and β :



Solving iteratively with Gauss-Seidel's policy would require 20 applications of equation fig.7.1.a plus 5 more applications to prove stabilization. However for the particular problem of live variables an optimal approximation sequence exists ([2],[33]). For example initializing live(b) by used(b) and applying the equation fig.7.1.a in the order b=1,2,3,4,3,2,1,0 we obtain :

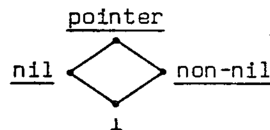
live(b) Steps \ b	0	1	2	3	4	5
Initialization	∅	{β}	∅	∅	{α}	∅
1		{β}				
2			{β}			
3				{β}		
4					{α}	
5				{α,β}		
6			{α,β}			
7		{α,β}				
8	∅					

Therefore 8 applications of the equation fig.7.1.a are strictly necessary (instead of 20) and no supplementary applications are needed to prove that the iteration process has converged (instead of 5).

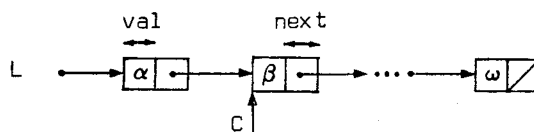
7.2 APPLICATION TO FINITE-INFINITE STATE ANALYSIS OF PROGRAMS

An analysis of programs usually performed by compilers is *type determination*. Compile-time *type verification* consists in verifying that the declarations of the program are respected in the instructions. Otherwise stated the programmer provides an exact or approximate solution to a system of type equations by means of declarations and the compiler simply verifies the correctness of this solution (4.4). On the contrary some programming languages have no declarations whereas in others the global declarations are not always sufficient for local type checking (see [14]) so that *type discovery* is necessary. Most modern programming languages permit programs using a potentially infinite number of types so that approximation is necessary (see e.g. (4.3.2), [15], [61]). Compile time analyses connected to type checking are control of correct access to data structures [56], finite state program testing [25], elimination of unnecessary copying operations in set languages [51], etc.

A classical example where global declarations are not sufficient for local type checking is the one of static correctness check of access to records via pointers. One must distinguish between nil and non-nil pointers. The global declarations must be locally refined by the compiler according to the following schema :



Consider the simple problem of finding the kth value in a linked linear list L :



A tentative solution may be the following :

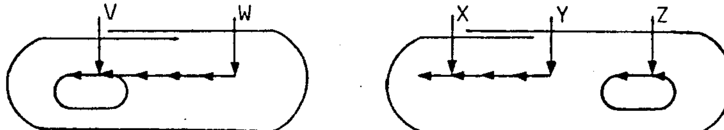
```
          if k≤0 then error fi;
{1}       C := L ;
          while k≠1 do
{2}          k := k-1;
{3}          if C=nil then error fi;
{4}          C := C↑.next;
{5}       od;
          ... C.val ...
```

Taking account of the fact that L may be an empty list at line {1} and that the function "next" delivers a nil or non-nil pointer the system of type equations is the following :

$$\begin{cases} C_1 = \underline{\text{pointer}} \\ C_2 = C_1 \sqcup C_4 \\ C_3 = C_2 \sqcap \underline{\text{non-nil}} \\ C_4 = \underline{\text{pointer}} \\ C_5 = C_1 \sqcup C_4 \end{cases}$$

The least solution shows that C_3=non-nil so that the access to a record via C at line 3 is correct. Yet C_5=pointer at line 5, and from this diagnosis the programmer should be able to discover that he has forgotten the case of a list of length k-1.

A problem which is frequently met when compiling programs involving pointers is that of determining whether two pointer variables may or not point directly or indirectly to the same record. To answer that question the compiler may analyze the program and locally partition the pointer variables into disjoint collections ([14],[30]). A collection is a set of pointer variables which may possibly point directly or indirectly to the same record. On the contrary two pointers belonging to disjoint collections cannot designate the same object. The two collections of the following example :



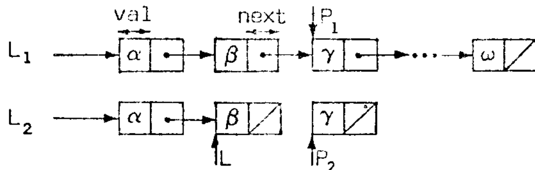
will be denoted by /V,W/X,Y,Z/. Note that collection /X,Y,Z/ is not minimal but /X,Y/Z/ is correctly approximated by /X,Y,Z/.

Let S_1 and S_2 be two sets of collections. The union $S_1 \sqcup S_2$ is a set of collections $/C_1/.../C_k/$ such that $X \in C_i$ and $Y \in C_i$ if and only if there exists a finite sequence $T_0=X, T_1, ..., T_r=Y$ of pointer variables such that for any element j of [1,r] there exists a collection C belonging to S_1 or S_2 such that $T_{j-1} \in C$ and $T_j \in C$. For example /A,B,C/D,E/ $\sqcup$ /F,A,G/H/ = /A,B,C,F,G/D,E/H/ since if on one hand A may point to a record referenced by B and C, and on the other hand, A may point to a record referenced by F and G, it is clear that A,B,C,F and G may point to the same record. Let us also define the extraction of a variable X from a set of collections $S = /C_1/.../X_1,X_2,...,X,...,X_n/.../C_m/$ by $\underline{\varepsilon}(X,S) = /C_1/.../X/X_1,X_2,...,X_n/.../C_m/$.

Let us now examine the association of a system of equations with any particular program. After the instructions "X:=nil" or "X:=Y" where Y is known to be nil or "if X=nil then ..." or "new(X)" it is known that X will point to no record at all or will be the only pointer to the newly allocated record. Thus we have isolated a collection (empty or consisting of a single record). The output set of collections S_0 corresponding to the input set of collections S_i is therefore $S_0=\underline{\varepsilon}(X,S_i)$. More generally, with an input set of collections S_i, a pointer assignment

such as "X↑.next...↑.next := Y↑.next...↑.next" where the selector parts are optional may cause X and Y to indirectly point to a common record. Hence they have to be put in the same collection. The output predicate will be $S_0 = S_I \sqcup /X,Y/$. Note that the above assignment performs a profound modification in the structure referenced by X and Y. This modification may cause a collection to be broken into two disjoint sub-collections. However since the internal organization of collections is not known it is impossible to observe the effect of this deep modification, except in the obvious cases "X:=Y" or "X :=Y↑.next...↑.next" which will cause X to be disconnected from its collection and be connected to a record of the collection of Y. When X and Y are not the same variable, the output set of collections S_0 will be related to the input set of collections S_I by $S_0 = \underline{\varepsilon}(X, S_I) \sqcup /X,Y/$.

Let us now consider an example which consists in copying a linked linear list :



the following procedure is supposed to do the job :

```
       procedure copy (L1:list; var L2 :list);
          var P1,P2,L:list;
       begin
{0}
{1}       P1:=L1; L2:=nil; L:=nil;
{2}       while P1≠nil do
{3}          new(P2); P2↑.val:=P1↑.val; P2↑.next:=nil;
{4}          if L=nil then
{5}             L2:=P2;
{6}          else
{7}             L↑.next:=P2;
{8}          fi;
{9}          L:=P2; P1 :=P1↑.next;
{10}      od;
       end;
```

According to our abstract interpretation of the basic constructs of the language we can now establish the corresponding system of equations :

$$
\begin{cases}
S_1 = \underline{\varepsilon}(L, \underline{\varepsilon}(L_2, \underline{\varepsilon}(P_1, S_0) \sqcup /P_1, L_1/)) \\
S_2 = S_1 \sqcup S_9 \\
S_3 = \underline{\varepsilon}(P_2, S_2) \\
S_4 = \underline{\varepsilon}(L, S_3) \\
S_5 = \underline{\varepsilon}(L_2, S_4) \sqcup /L_2, P_2/ \\
S_6 = S_3 \\
S_7 = S_6 \sqcup /L, P_2/ \\
S_8 = S_5 \sqcup S_7 \\
S_9 = \underline{\varepsilon}(L, S_8) \sqcup /L, P_2/ \\
S_{10} = \underline{\varepsilon}(P_1, S_1 \sqcup S_9)
\end{cases}
$$

For equations 2 and 6 the fact that $P_1 \neq$ nil or L≠nil gives no information on collections. In equation 3 the assignment of non-pointer values "P_2↑.val:=P_1↑.val" and a deep modification "P_2↑.next:=nil" in the structure pointed to by P_2 are ignored. For equation 9 notice that the statement "P_1 :=P_1 ↑.next" leaves P_1 in the same collection.

The system of equations can be solved by successive approximations since the number of collections constructed on a finite set of pointer variables is finite. Since we want the collections to be as refined as possible we compute the least solution starting from the infimum $/L_1/L_2/P_1/P_2/L/$. For the entry condition S_0 in the procedure we start with the most

disadvantageous initial situation where on the one hand the parameters $/L_1,L_2/$ and on the other hand the local variables $/P_1,P_2,L/$ are supposed to be in the same collection.

Initialization :

$$\begin{cases} S_0^0 = /L_1,L_2/P_1,P_2,L/ \\ S_i^0 = /L_1/L_2/P_1/P_2/L/,\ i=1..10 \end{cases}$$

Step 1 :

$$\begin{cases} S_1^1 = \underline{\varepsilon}(L,\underline{\varepsilon}(L_2,\underline{\varepsilon}(P_1,/L_1,L_2/P_1,P_2,L/) \sqcup /P_1,L_1/)) = \underline{\varepsilon}(L,\underline{\varepsilon}(L_2,/L_1,L_2/P_1/P_2,L/ \sqcup /P_1,L_1/)) \\ \quad = \underline{\varepsilon}(L,\underline{\varepsilon}(L_2,/L_1,L_2,P_1/P_2,L/)) = /L_1,P_1/L_2/P_2/L/ \\ S_2^1 = /L_1,P_1/L_2/P_2/L/ \sqcup /L_1/L_2/P_1/P_2/L/ = /L_1,P_1/L_2/P_2/L/ \\ S_3^1 = \underline{\varepsilon}(P_2,/L_1,P_1/L_2/P_2/L/) = /L_1,P_1/L_2/P_2/L/ \\ S_4^1 = \underline{\varepsilon}(L,/L_1,P_1/L_2/P_2/L/) = /L_1,P_1/L_2/P_2/L/ \\ S_5^1 = \underline{\varepsilon}(L_2,/L_1,P_1/L_2/P_2/L/) \sqcup /L_2,P_2/ = /L_1,P_1/L_2,P_2/L/ \\ S_6^1 = /L_1,P_1/L_2/P_2/L/ \\ S_7^1 = /L_1,P_1/L_2/P_2/L/ \sqcup /L,P_2/ = /L_1,P_1/L_2/P_2,L/ \\ S_8^1 = /L_1,P_1/L_2,P_2/L/ \sqcup /L_1,P_1/L_2/P_2,L/ = /L_1,P_1/L_2,P_2,L/ \\ S_9^1 = \underline{\varepsilon}(L,/L_1,P_1/L_2,P_2,L/) \sqcup /L,P_2/ = /L_1,P_1/L_2,P_2,L/ \end{cases}$$

Cycling in the while-loop until the invariants $S_1,\ldots,S_9$ have stabilized we go on step 2 :

$$\begin{cases} S_2^2 = /L_1,P_1/L_2/P_2/L/ \sqcup /L_1,P_1/L_2,P_2,L/ = /L_1,P_1/L_2,P_2,L/ \\ S_3^2 = \underline{\varepsilon}(P_2,/L_1,P_1/L_2,P_2,L/) = /L_1,P_1/L_2,L/P_2/ \\ S_4^2 = \underline{\varepsilon}(L,/L_1,P_1/L_2,L/P_2/) = /L_1,P_1/L_2/P_2/L/ \\ S_5^2 = \underline{\varepsilon}(L_2,/L_1,P_1/L_2/P_2/L/) \sqcup /L_2,P_2/ = /L_1,P_1/L_2,P_2/L/ \\ S_6^2 = /L_1,P_1/L_2,L/P_2/ \\ S_7^2 = /L_1,P_1/L_2,L/P_2/ \sqcup /L,P_2/ = /L_1,P_1/L_2,L,P_2/ \\ S_8^2 = /L_1,P_1/L_2,P_2/L/ \sqcup /L_1,P_1/L_2,P_2,L/ = /L_1,P_1/L_2,P_2,L/ \\ S_9^2 = \underline{\varepsilon}(L,/L_1,P_1/L_2,P_2,L/) \sqcup /L,P_2/ = /L_1,P_1/L_2,P_2,L/ \\ S_{10}^2 = \underline{\varepsilon}(P_1,\ /L_1,P_1/L_2/P_2/L/ \sqcup /L_1,P_1/L_2,P_2,L/) = /L_1/P_1/L_2,P_2,L/ \end{cases}$$

The iterates have stabilized and the main result is that although L_1 and L_2 may share records on entry of the procedure "copy" ($S_0=/L_1,L_2/P_1,P_2,L/$) it is guaranteed that this is not the case on exit of the procedure ($S_{10}=/L_1/P_1/L_2,P_2,L/$). The local collections may be used by compilers in several ways. An optimizing compiler will be able to limit the number of objects which are supposed to have been modified by side-effects when assigning to objects designated by pointers (which is useful in register allocation). The compiler may insert a call to the garbage collector in the code when no variable in a collection is live (i.e. all variables in the collection are not used before being assigned to). Also run-time tests may be inserted before a statement "dispose(X)" to verify that no live variable in the collection of X may access the record referenced by X which will be returned to the free storage.

8. CONCLUSION

Numerous methods have been used for determining properties of programs and we think that our fixed point approach in the semantic analysis of programs provides a convenient framework for expressing the deep unity underlying apparently unrelated techniques such as denotational semantics, logical analysis of programs, program performance prediction, data flow analysis, extended type discovery, etc. By fixed point approach in the semantic analysis of programs we refer to the whole of techniques for determining program properties which take as starting

point the fact that these properties can be defined as the least solution to a system of equations which is associated with the program by approximating the formal semantic definition of this program.

In our presentation we have basically considered the semantic analysis of sequential programs. We also have to consider the problems of applying this approach to less restricted programming languages where the concepts of "program point" and "identifier" are more complex. Nested scopes and recursion are treated in [15] but more complicated language features must be studied such as for example procedures as parameters or jumps out of procedures.

The general problem of finding an algorithm to solve the equations associated with any sequential program is undecidable. In practice the solution to these equations can be approximated and we presented fixed point approximation algorithms. Considerable progress can be made with regard to efficiency of these computation methods and mainly with regard to the preciseness of the approximation. By comparison with numerical equations it is clear that the problem of approximate resolution of fixed point equations in infinite lattices has not been studied in depth and is a promising research area.

Program optimization and data flow analysis is one area where fixed point computation algorithms have been intensively developed. However these algorithms are often expressed in graphical terms and devoted to specific applications where lattices are of finite length. It is certainly interesting to forget the specific applications and express these data flow analysis algorithms as fixed point computation methods. Also it is clear that the ability to consider arbitrary lattices would enlarge the scope of data flow analysis techniques.

The area of compiler writing and compiler correctness is one natural application. In compilers the phases of syntax analysis and verification of context conditions have been suitably formalized by context-free grammars and attribute grammars. The well-known advantages of these formalisms are their understandability coming from their declarative aspect as well as the fact that they automatically lead to constructive algorithms. One find again the same advantages in our fixed point formalization of the subsequent phase of compile-time analysis of programs. The specification of the abstract space of approximate properties and the definition of the rules for associating a system of equations with a program have a descriptive or declarative aspect. (At that time compiler correctness can be established by proving the validity of these rules with respect to the semantics of the language). The constructive aspect follows from the existence of fixed point generation or approximation algorithms. (Note also that attribute grammars associate a system of fixed point equations with a syntactic tree whereas we associate a system of equations with a control graph).

Finally in the area of logical analysis of programs where one has to deal with undecidable problems the notion of approximation is unvoidably essential. One can imagine to design a wide spectrum of approximate properties or approximation heuristics to cope with analysis of usual programs. However it is clear that some specific programs will need peculiar analyses which cannot be foreseen in advance. Hence it is essential to design languages allowing the user to define and eventually guide such analyses. The present research on abstract data types seems to be the first steps in that direction.

Acknowledgements : We were very lucky to have Mrs. F. Blanc and Mrs. C. Puech do the typing for us.

REFERENCES

1. Abian, S. and Brown, A.B. A theorem on partially ordered sets with applications to fixed point theorems. *Canad. J. Math.* 13(1961), 78-82.

2. Aho, A.V., and Ullman, J.D. Node listings for reducible flow graphs. Proc. 7th Annual ACM Symp. on Theory of Computing, May 1975, 177-185.

3. Allen, F.E. A basis for program optimization. Proc. IFIP Cong. 71, Vol. 1, North-Holland Pub. Co., Amsterdam, 1971, 385-390.

4. Baudet, G. Asynchronous iterative methods for multiprocessors. Res. Rep., Computer Sci. Dept., Carnegie Mellon U., Nov. 1976.

5. Birkhoff, G., *Lattice Theory,* AMS Coll. Pub., XXV, 3rd ed., Providence, R.I., 1967.

6. Burstall, R.M. Proving properties of programs by structural induction. *Computer Journal,* Vol. 12, 1969, 41-48.

7. Burstall, R.M. Program proving as hand simulation with a little induction. Proc. IFIP Cong. 74, Software, North-Holland, Pub. Co., Amsterdam, 1974, 308-312.

8. Cheatham, T.E., and Townley, J.A. Symbolic evaluation of programs : a look at loop analysis Proc. of the 1976 ACM Symp. on Symbolic and Algebraic Computation, Aug. 1976.

9. Cocke, J. Global common subexpression elimination. *SIGPLAN Notices 5,* 7(July 1970), 20-24.

10. Cohen, J., and Katcoff, J. Symbolic solution of finite difference equations, R.R., Physics Dept., Brandeis U., Waltham Mass., July 1976.

11. Courcelle, B., and Nivat, M. Algebraic families of interpretations. Proc. 17th Symp. on Foundations of Computer Sci., Houston, Oct. 1976.

12. Cousot, P., and Cousot, R. Static determination of dynamic properties of programs. Proc. 2nd Int. Symp. on Programming, B. Robinet (Ed.), Dunod, Paris, April 1976. [Also in *MOHL Bulletin, No.5,* P. Cousot (Ed.), IRIA, Rocquencourt, France, (Sept. 1976), 27-52].

13. Cousot, P., and Cousot, R. Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints. Conf. Rec. of the 4th ACM Symp. on Principles of Programming Languages, Los Angeles, Calif., Jan. 1977, 238-252.

14. Cousot, P., and Cousot, R. Static determination of dynamic properties of generalized type unions. ACM. Conf. on Language Design for Reliable Software, Raleigh, North Carolina, March 1977. *SIGPLAN Notices 12,* 3(March 1977), 77-94.

15. Cousot, P., and Cousot, R. Static determination of dynamic properties of recursive procedures. IFIP W.G.2.2. Working Conf. on Formal Description of Programming Concepts, St. Andrews, New Brunswick, Canada, Aug. 1977.

16. Cousot, P., and Cousot, R. Automatic synthesis of optimal invariant assertions : mathematical foundations. Proc. of the ACM Symposium on Artificial Intelligence & Programming Languages, Rochester, New York, Aug. 1977.

17. De Bakker, J.W., and Scott, D. A theory of programs. Unpublished Notes, IBM Seminar, Vienna, 1969.

18. Dijkstra, E.W. *A discipline of programming.* Prentice-Hall, Englewood Cliffs, N.J., 1976.

19. Floyd, R.W. Assigning meaning to programs. Proc. Symp. in Appl. Math., Vol. 19, J.T. Schwartz (Ed.), Amer. Math. Soc., Providence, R.I., 1967, 19-32.

20. Goguen, J.A., Thatcher, J.W., Wagner, E.G., and Wright, J.B. Initial algebra semantics and continuous algebras. *JACM 24,* 1(Jan. 1977).

21. Grief, I., and Waldinger, R.J. A more mechanical approach to program verification. Proc. 1st Int. Symp. on Programming, B. Robinet (Ed.). Lecture Notes in Computer Sci., Springer-Verlag, Berlin, April 1974, 109-118.

22. Hantler, S.L., and King, J.C. An introduction to proving the correctness of programs. *Computing Surveys 8,* 3(Sept. 1976), 331-353.

23. Hecht, M.S., and Ullman, J.D. Analysis of a simple algorithm for global flow problems. Conf. Rec. of the ACM Symp. on Principles of Programming Languages, Boston, Mass., Oct.1973, 207-217.

24. Hehner, E.C.R. do considered od : a contribution to the programming calculus. Tech. Rep. CSRG-75, Computer Systems Research Group, U. of Toronto, Nov. 1976.

25. Henderson, P. Finite state modelling in program development. Proc. Int. Conf. on Reliable Software, Los Angeles, Calif., April 1975, 221-227.

26. Hitchcock, P., and Park, D. Induction rules and proofs of termination. Proc. IRIA Symp. Automata, Languages and Programming, North-Holland Pub. Co., July 1972, 225-251.

27. Hoare, C.A.R. An axiomatic basis of computer programming. *Comm. ACM 12,* 10(Oct. 1969), 576-580.

28. Höft, H., and Höft, M. Some fixed point theorems for partially ordered sets. *Canad. J. Math. 28,* 5(1976), 992-997.

29. Kam, J.B., and Ullman, J.D. Monotone data flow analysis frameworks. *Acta Informatica 7,* 1977, 305-317.

30. Karr, M. Gathering information about programs. Mass. Computer Associates, Inc., CA-7507-1411, July, 1975.

31. Karr, M. Affine relationships among variables of a program. *Acta Informatica 6,* 1976, 133-151.

32. Katz, S., and Manna, Z. Logical analysis of programs. *Comm. ACM 19,* 4(April 1976), 188-206.

33. Kennedy, K. Node listings applied to data flow analysis. Conf. Rec. of the 2nd ACM Symp. on Principles of Programming Languages, Palo Alto, Calif., Jan. 1975, 10-21.

34. Kennedy, K. A comparison of two algorithms for global data flow analysis. *SIAM J. Computing 5,* 1(March 1976), 158-180.

35. Kennedy, K., and Zucconi, L. Applications of a graph grammar for program control flow analysis. Conf. Record. of the 4th ACM Symp. on Principles of Programming Languages, Los Angeles, Calif., Jan. 1977, 72-85.

36. Kildall, G.A. A unified approach to global program optimization. Conf. Rec. of the ACM Symp. on Principles of Programming Languages, Boston, Mass., Oct. 1973, 194-206.

37. King, J. A program verifier. Ph.D. Th., Dept. of Computer Sc., Carnegie-Mellon U., Pittsburgh, Pa., 1969.

38. King, J.C. Symbolic execution and program testing. *Comm. ACM 19,* 7(July 1976), 385-394.

39. Kleene, S.C. *Introduction to Metamathematics.* North-Holland Pub. Co., Amsterdam, 1952, 348-349.

40. Knaster, B. Un théorème sur les fonctions d'ensembles, *Ann. Soc. Polon. Math,* 6(1928), 133-134.

41. MacCarthy, J. A basis for a mathematical theory of computation. Computer Programming and Formal Systems, Braffort and Hirshberg (Eds.), North-Holland, Amsterdam, 1963, 33-69.

42. MacNeille, H.M. Partially ordered sets. *Trans. Amer. Math. Soc. 42,* (1937), 416-460.

43. Manna, Z. *Mathematical Theory of Computation.* Mc Graw-Hill, New York, 1974.

44. Manna, Z., Ness, S. and Vuillemin, J. Inductive methods for proving properties of programs. *Comm. ACM 16,* 8(Aug. 1973), 491-502.

45. Morris, J.H. Another recursion induction principle. *Comm. ACM 14,* 5(May 1971), 351-354.

46. Naur, P. Checking of operand types in ALGOL compilers. *BIT 5,* (1966), 151-163.

47. Nivat, M. On the interpretation of recursive program schemes. Symppsia Mathematica, Vol. XV Instituto Nazionale di Alta Mathematica, Italy, 1975, 255-281.

48. Park, D. Fixpoint induction and proofs of program properties. *Machine Intelligence 5,* B. Meltzer and D. Michie (Eds.), Edinburgh U. Press, 1969, 59-78.

49. PASCAL. Wirth, N. The programming language PASCAL, *Acta Informatica 1*, 1(1971), 35-63.

50. Robert, F. Sur la transformation de Gauss-Seidel. Séminaire d'Analyse Numérique, No. 255, Mathématiques Appliquées, U.S.M.G., Grenoble, Oct. 1976.

51. Schwartz, J.T. Automatic data structure choice in a language of very high level. *Comm. ACM 18*, 12(Dec. 1975), 722-728.

52. Scott, D. Outline of a mathematical theory of computation. Proc. of the 4th Ann. Princeton Conf. on Information Sciences and Systems, Princeton, 1970, 169-176.

53. Scott, D. Data types as lattices. *SIAM J. Computing 5*, 3(Sept. 1976), 522-587.

54. Scott, D., and Strachey, C. Towards a mathematical semantics for computer languages. Proc. Symp. on Computers and Automata, Polytechnic Inst. of Brooklyn, Vol. 21, 1971, 19-46.

55. SELECT. Boyer, R.S., Elspas, B., and Levitt, K.N. SELECT - A formal system for testing and debugging programs by symbolic execution. Proc. Int. Conf. on Reliable Software, Los Angeles, Calif., April 1975, 234-245.

56. Sintzoff, M. Calculating properties of programs by valuations on specific models. Proc. ACM Conf. on Proving Assertions about Programs. *SIGPLAN Notices 7*, 1(1972), 203-207.

57. Sintzoff, M. Vérification d'assertions pour des fonctions utilisables comme valeurs et affectant des variables extérieures. Proc. Int. Symp. on Proving and Improving Programs, Arc et Senans, France, July 1975, 11-27.

58. Sintzoff, M. Iterative methods for the generation of successful programs. M.B.L.E. Research Lab., Brussels, 1977.

59. Tarjan, R.E. Iterative algorithms for global flow analysis. Tech. Rep. CS 76-545, Comp. Sc. Dept., Stanford U., Feb. 1976.

60. Tarski, A. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Math. 5*, (1955), 285-309.

61. Tennenbaum, A. Type determination for very high level languages. NSO-3, Courant Inst. of Math. Sci., New-York U., Oct. 1974.

62. Ullman, J.D. Data flow analysis. Proc. 2nd USA-Japan Computer Conf., Montvale, N.J., AFIPS Press, 1975.

63. Vuillemin, J. Proof techniques for recursive programs. STAN-CS-73-393, Computer Science Dept., Stanford U., Oct. 1973.

64. Wegbreit, B. Mechanical program analysis. *Comm. ACM 18*, 9(Sept. 1975), 528-539.

65. Wegbreit, B. Property extraction in well-founded property sets. *I.E.E.E. Trans. on Soft. Eng.*, Vol. SE-1, No 3, (Sept. 1975), 270-285.

66. Wegbreit, B. Verifying program performance. *JACM 23*, 4(Oct. 1976), 691-699.

67. Yonezawa, A. Symbolic-evaluation as an aid to program synthesis. Working paper 124, Artificial Intelligence Lab., Mass. Inst. of Technology, April 1976.