TOWARDS A UNIVERSAL MODEL

FOR

STATIC ANALYSIS OF PROGRAMS

(PRELIMINARY DRAFT)
January 1977

Patrick COUSOT* and Radhia COUSOT**

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

TOWARDS A UNIVERSAL MODEL
FOR
STATIC ANALYSIS OF PROGRAMS

(PRELIMINARY DRAFT)
January 1977

Patrick COUSOT* and Radhia COUSOT**

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

ABSTRACT

The *abstract interpretation* of programs is a mathematical model for static analysis of programs :

a new interpretation is given to the program text and this allows the building of a system of equations. The analysis of the program then consists in verifying that a solution provided by the user is correct, or in computing this solution, or else in discovering a good approximation of this solution.

---

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

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

The abstract interpretation of programs was formalized in Cousot [1977a]. Only a few lattice theoretical results are recalled here for the paper to be self-contained.

Our main purpose is to show that most program analysis techniques may be understood and modelled as abstract interpretations of programs. This is illustrated by the following examples :

- Global data flow analysis,
- Analysis of program performance,
- Denotational semantics of programs,
- Proofs of program partial correctness,
- Proofs of program termination,
- Symbolic execution of programs,
- Type verification/discovery,
- Finite/infinite state program analysis.

The application of abstract interpretation in studying these apparently unrelated techniques renders the connexion between them clear and straightforward. Moreover, numerous mis-understood problems are highlighted when explained by the model of abstract interpretation.

1. INTRODUCTION

Let us introduce the abstract interpretation of programs by means of a very intuitive and trivial example. Suppose we are interested in discovering the sign of the integer variable i in the (non-terminating) program :

i := 1 ; while true do i := i+1 ;

Let us note $\dot{+}$ the fact that i is positive, $\dot{-}$ that i is negative, $\dot{\pm}$ the fact that i is an integer which sign is unknown. For reasons which will be explicited later we note $\perp$ the fact that nothing is known about i.

Besides we associate invariants P_1, P_2, P_3 with various points {1}, {2} and {3} of the program :

i := 1 {1} ; while true do {2} i := i+1 {3} ;

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

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

- i is positive at program point {1} since it is equal to 1. Therefore :

$$P_1 = \dot{+}$$

- The sign of i at point {2} may be P_1 when coming from point {1} or P_3 when coming from point {3}. Therefore

$$P_2 = P_1 \underline{\text{ or }} P_3$$

where the operator or is defined by :

$$\dot{+} \underline{\text{ or }} \dot{+} = \dot{+}$$
$$\dot{+} \underline{\text{ or }} \dot{-} = \dot{\pm}$$
$$\dot{+} \underline{\text{ or }} \dot{\pm} = \dot{\pm}$$
$$\dot{+} \underline{\text{ or }} \perp = \dot{+}$$
etc.

- Finally suppose the sign of i is P_2, the sign of "i+1" is "$P_2 \oplus \dot{+}$", where the operator $\oplus$ is defined by the rules of signs :

$$\dot{+} \oplus \dot{+} = \dot{+}$$
$$\dot{-} \oplus \dot{+} = \dot{\pm}$$
$$\dot{\pm} \oplus \dot{+} = \dot{\pm}$$
$$\perp \oplus \dot{+} = \perp$$
etc.

Hence the sign P_3 of i after the assignement "i := i+1" is "$P_2 \oplus \dot{+}$". Therefore

$$P_3 = P_2 \oplus \dot{+}$$

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

$$\begin{cases} P_1 = \dot{+} \\ P_2 = P_1 \underline{\text{ or }} P_3 \\ P_3 = P_2 \oplus \dot{+} \end{cases}$$

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

$$\begin{cases} X_1 = \dot{+} \\ X_2 = X_1 \underline{\text{ or }} X_3 \\ X_3 = X_2 \oplus \dot{+} \end{cases}$$

would be an acceptable candidate for the invariants.

Solutions of that system exist, and in general are not unique :

	X_1	X_2	X_3
(a)	$\dot{+}$	$\dot{+}$	$\dot{+}$
(b)	$\dot{+}$	$\dot{\pm}$	$\dot{\pm}$

However, a best solution (a) exists since $\dot{+}$ is a more precise result than $\dot{\pm}$.

The best solution (a) can be automatically constructed by successive approximations, as follows :

First approximation :

$$\begin{cases} X_1 = \perp \\ X_2 = \perp \\ X_3 = \perp \end{cases}$$

The second approximation is obtained by replacing X_1, X_2, X_3 by $\perp$ in the right hand side of the system of equations. We get :

$$\begin{cases} X_1 = \dot{+} \\ X_2 = \perp \ \underline{\text{or}} \ \perp = \perp \\ X_3 = \perp \ \boxplus \ \dot{+} = \perp \end{cases}$$

Third approximation :

$$\begin{cases} X_1 = \dot{+} \\ X_2 = \dot{+} \ \underline{\text{or}} \ \perp = \dot{+} \\ X_3 = \perp \ \boxplus \ \dot{+} = \perp \end{cases}$$

Fourth approximation :

$$\begin{cases} X_1 = \dot{+} \\ X_2 = \dot{+} \ \underline{\text{or}} \ \perp = \dot{+} \\ X_3 = \dot{+} \ \boxplus \ \dot{+} = \dot{+} \end{cases}$$

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

$$\begin{cases} X_1 = \dot{+} \\ X_2 = \dot{+} \ \underline{\text{or}} \ \dot{+} = \dot{+} \\ X_3 = \dot{+} \ \boxplus \ \dot{+} = \dot{+} \end{cases}$$

This is a certainly toilsome but at least systematic and simply automatizable way to prove that i is positive in the program :

i := 1 {1} ; while true do {2} i := i+1 {3} ;

However this abstract interpretation is not very powerful since changing "i := i+1" by "i := i-2", we get a modified system of equations :

$$\begin{cases} X_1 = \dot{+} \\ X_2 = X_1 \ \underline{\text{or}} \ X_3 \\ X_3 = X_2 \ \ominus \ \dot{+} \end{cases}$$

which least solution is :

X_1	X_2	X_3
$\dot{+}$	$\dot{\pm}$	$\dot{\pm}$

The fact that i is negative at program point {3} is not captured by this abstract interpretation because of the rule "$\dot{+} \ominus \dot{+} = \dot{\pm}$". A more careful analysis taking account of the absolute value of i would be necessary to discover this fact.

The same inaccuracy occurs with other abstract interpretations (such as casting out of nines in arithmetic, parity checks in hardware, dimensional analysis in physics, ...) : they permit to automatically verify sufficient (yet in general not necessary) conditions of truth or falseness of a property, (Sintzoff[1972]).

However more refined (but may be not fully automatizable) abstract interpretations may be used to analyze stronger properties of programs. For example the program

i := 1 {1} while i < N do {2} i := i+1 {3} ;

may be analyzed using the following equations over predicates :

$$\begin{cases} P_1 = (i = 1) \\ P_2 = (P_1 \ \underline{\text{or}} \ P_3) \ \underline{\text{and}} \ (i < N) \\ P_3 = P_2(i \leftarrow i-1) \end{cases}$$

which solution is :

$$\begin{cases} P_1 = (i = 1) \\ P_2 = (1 < N)\underline{and}(1 \leq i)\underline{and}(i < N) \\ P_3 = (1 < N)\underline{and}(2 \leq i)\underline{and}(i \leq N) \end{cases}$$

The purpose of this paper is to show that most techniques for analyzing properties of programs may be understood as particular abstract interpretations.

2. INTRODUCTION TO THE MATHEMATICAL MODEL USED IN ABSTRACT INTERPRETATION OF PROGRAMS

2.1 *The Complete Lattice of Abstract Contexts*

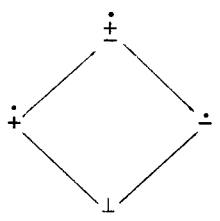
The set A of abstract contexts is supposed to be a lattice. We use the symbols :

$$\sqsubseteq,\ \bot,\ \top,\ \sqcup,\ \sqcap,\ \bigsqcup,\ \bigsqcap$$

to denote respectively the partial ordering, the infimum, the supremum, the join of two elements, the meet of two elements, the join of a set of elements, and the meet of a set of elements. The definitions and mathematical properties of these notions can be found in many places, for example Birkhoff[1973].

Example

In the introduction we used $A = \{\bot, \dot{+}, \dot{-}, \dot{\pm}\}$ where $\top$ was denoted $\dot{\pm}$. The ordering relation $\sqsubseteq$ is defined by its diagram :



For instance $\dot{+} \sqsubseteq \dot{\pm}$ since the assertion that i is a positive integer is less unprecise than the assertion that i is simply an integer. The introduction of $\perp$ is necessary to make the poset $\{\dot{+}, \dot{-}, \dot{\pm}\}$ a lattice.

In general we will consider infinite lattices and will require them to be complete, that is by definition for any non-void part $\{X_i \mid i \in \Delta\}$ of A the join :

$$\bigsqcup\{X_i \mid i \in \Delta\}$$

and meet :

$$\bigsqcap\{X_i \mid i \in \Delta\}$$

must exist for any arbitrary indexing set Δ.

Example

Let A be the subset of rational numbers included in the interval $[3, 4]$. It is a lattice with ordering $\leq$, infimum 3, supremum 4, join max, meet min. The strictly increasing chain $P_1 = 3$, $P_2 = 3.1$, $P_3 = 3.14$, ..., $P_6 = 3.14159$, ... converges towards π which does not belong to A. Consequently $\bigsqcup\{P_i \mid i \geq 1\}$ does not exist in A which proves that A is not complete.

Hypothesis

The set A of abstract contexts is supposed to be a complete lattice $(\sqsubseteq, \perp, \top, \sqcup, \sqcap, \bigsqcup, \bigsqcap)$.

Lemma

The set A^k is a complete lattice $(\sqsubseteq_k, \perp_k, \top^k, \sqcup_k, \sqcap_k, \bigsqcup_k, \bigsqcap_k)$.

The following definitions are usual :

$$\{\langle X_1, \ldots, X_k\rangle \sqsubseteq_k \langle Y_1, \ldots, Y_k\rangle\} \Longleftrightarrow \{(X_i \sqsubseteq Y_i), \ \forall i = 1, \ldots, k\}$$

$$\langle X_1, \ldots, X_k\rangle \sqcup_k \langle Y_1, \ldots, Y_k\rangle = \langle X_1 \sqcup Y_1, \ldots, X_k \sqcup Y_k\rangle$$

etc.

We drop the subscript k when unambiguously available from context.

2.2 *System of Equations Associated with a Program*

A fixpoint equation :

$$X = F(X)$$

with k variables, that is of the form

$$\begin{cases} X_1 = f_1(X_1, \ldots, X_k) \\ \vdots \\ X_k = f_k(X_1, \ldots, X_k) \end{cases}$$

is associated to the k cutpoints of the program.
The variables $(X_1, \ldots, X_k)$ range over the complete lattice A.

The syntactic process which permits to associate a system of equations with a particular program is not formalized here, and will be informally sketched for each of the further examples.

Definition

A function $f : D \to D'$ from the poset $(D, \leq)$ in the poset $(D', \sqsubseteq)$ is order-preserving (synonymously monotonic or isotone) if and only if :

$$\{\forall (x, y) \in D^2, \{x \leq y\} \Longrightarrow \{f(x) \sqsubseteq f(y)\}\}$$

Hypothesis

The functions $f_i : A^k \to A$ are supposed to be order-preserving.
(The point of monotonicity is that the better we define the argument contexts $(X_1, \ldots, X_k)$, the better we define the resulting context $f_i(X_1, \ldots, X_k)$).

Lemma

The map $F : A^k \to A^k$ is preserving the ordering $\sqsubseteq_k$.

2.3 *Existence of Solutions to the System of Equations*

Theorem (Tarski[1955]).

The order-preserving map F of the complete lattice A^k in itself has fixpoints (that is $F(a) = a$ for some $a \in A^k$). In fact, the fixpoints of F form a complete lattice for ordering $\sqsubseteq_k$.

Corollary 1 (Least fixpoint).

The least fixpoint $\mu(F)$ of F is given by :

$$\mu(F) = \Pi_k \{X \in A^k \mid F(X) \sqsubseteq_k X\}$$

Corollary 2 (Greatest fixpoint).

The greatest fixpoint $M(F)$ of F is given by the formula :

$$M(F) = \sqcup_k \{X \in A^k \mid X \sqsubseteq_k F(X)\}$$

The above results permit to discuss the existence of solutions to the system of equations. However they do not provide an algorithmic construction of the extreme fixpoints. Additional hypothesis are necessary to provide constructive methods.

2.4 *Construction of the Extreme Solutions of the System of Equations*

2.4.1 Construction of the Least Fixpoint of F

Definition

A map $f : D \to D'$ from the complete lattice $(D, \sqsubseteq, \sqcup)$ in the complete lattice $(D', \sqsubseteq', \sqcup')$ is called *upper-semi-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

$$f(\sqcup X) = \sqcup' \{f(x) \mid x \in X\}$$

Note that upper-semi- continuous functions have the monotonicity property and preserve upper limits.

$$f(\lim_{n\to\infty} (x_n)) = \lim_{n\to\infty} (f(x_n))$$

Hypothesis

The functions $f_i : A^k \to A$ are supposed to be upper-semi-continuous, i = 1..k.

Lemma

The map $F : A^k \to A^k$ is an upper-semi-continuous function from the complete lattice A^k in itself.

Theorem (Kleene[1952], first recursion theorem)

The least fixpoint $\mu(F)$ of an upper-semi-continuous function F from the complete lattice A^k in itself is the limit of *the increasing approximation sequence* given by the formula :

$$\mu(F) = \bigsqcup_{n=0}^{\infty} F^n(\bot^k)$$

where

F^n is the n-fold composition of F with itself ($F^0(X) = X$, $F^{n+1}(X) = F(F^n(X))$) and $\bot^k$ is the least element of A^k (a k-tuple which elements are equal to $\bot$).

Notice that :

$$\bot^k \sqsubseteq_k F(\bot^k)$$

since $\bot^k$ is the infimum of the lattice A^k. Hence

$$F^n(\bot^k) \sqsubseteq_k F^{n+1}(\bot^k)$$

is easy to prove by recurrence on n using the property that F is order preserving. This implies that the successive approximations :

$$\bot^k = F^0(\bot^k),\ F^1(\bot^k),\ \ldots,\ F^n(\bot^k),\ \ldots$$

form an increasing chain :

$$\bot^k = F^0(\bot^k) \sqsubseteq_k F^1(\bot^k) \sqsubseteq_k \ldots \sqsubseteq_k F^n(\bot^k) \sqsubseteq_k \ldots$$

Hence

$$\bigsqcup_{n=0}^{m} F^n(\bot^k) = F^m(\bot^k)$$

and

$$\bigsqcup_{n=0}^{\infty} F^n(\bot^k) = \lim_{n\to\infty} F^n(\bot^k)$$

Infinitely many successive approximations may be necessary to pass to the limit when the approximation sequence is indefinitely strictly increasing.

Otherwise, the first p terms of the sequence are strictly increasing but terms of higher rank are equal to $F^p(\perp^k)$. This situation of utmost practical interest is found in particular when the lattice A^k (or simply A) satisfies the ascending chain condition.

Definition (ascending chain condition)

A partly ordered set (P, ≤) satisfies the ascending chain condition (resp. decreasing chain condition) if and only if every strictly increasing (resp. decreasing) chain is finite.

2.4.2 *Construction of the Greatest Fixpoint of F.*

Similarily, supposing the f_i, $i \in [1, k]$ to be lower-semi-continuous functions from the complete lattice A^k in A, the greatest fixpoint M(F) of F is the limit of *the decreasing approximation sequence* given by the formula :

$$M(F) = \prod_{n=0}^{\infty} F^n(\top^k)$$

2.4.3 *Extreme Fixpoints of F.*

The functions f_i, $i \in [1, k]$ will be said to be continuous when both lower-semi-continuous and upper-semi-continuous. In that case, the extreme fixpoints of F may be constructed as the limits of the increasing and decreasing approximation sequences.

2.5 *Discussion on Alternate Mathematical Models*

Arbitrary posets are not in general complete lattices. Other well-known fixpoint theorems might be used in such a case (a.o. Abian and Brown [1961], Höft[1976], etc.). Other convenient algebras permit to give constructive definitions of fixpoints (a.o. chain complete partly ordered sets, complete ordered F-magma Nivat[1974], Courcelle and Nivat[1976], initial continuous algebras Goguen et al.[1977]). 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 (a.o., Mac Neille[1937]).

2.6 *Warning*

Subsequently we will apply the above theorems without paying careful attention to formalities.

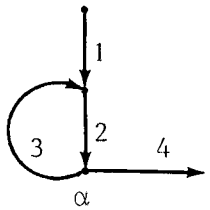
3. APPLICATION TO PERFORMANCE ANALYSIS OF PROGRAMS

This example of application is presented first, since it permits to understand the utilized mathematical model by analogy with numerical analysis techniques.

3.1 *Associating a System of Equations to a Program*

The performance of programs may be analyzed by deriving for each program point the final value of an imaginary counter which is incremented each time control goes through that point.

We will model the program by a directed graph which 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. We wish to determine the expected frequency C_i of traversing each arc i in the program during a single execution of the whole program. Under simplifying Markov assumptions (Cocke and Kennedy[1974]) the expected frequencies are given by the solution of a system of equations generated from the program by application of Kirschoff's laws of conservation of flow :

$$\begin{cases} C_1 = 1 & \text{(single entry arc)} \\ C_2 = C_1 + C_3 \\ C_3 = C_2 \cdot p(\alpha) \\ C_4 = C_2 \cdot (1 - p(\alpha)) \end{cases}$$

The main difficulty is to obtain the probabilities $p(\alpha)$. In general, an exact expression in terms of known properties of the input cannot be obtained, e.g. 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 independant of prior history. The values of these probabilities are supposed to be "given", e.g. might be determined by measurements.

3.2 *The Complete Lattice of Abstract Contexts*

Abstract contexts are positive reals $\mathbb{R}^+$ ordered by the natural ordering $\leq$ ($\subseteq$). The least upper bound operation $\sqcup$ is the maximum max, and the greatest lower bound operation $\sqcap$ is the minimum min operation. The infimum $\perp$ of the poset $\mathbb{R}^+$ is equal to $0 = \underline{MIN}\{i \mid i \in \mathbb{R}^+\}$. Notice that we are not considering a complete lattice since the expression

$$\underline{MAX}\{i \mid i \in \mathbb{R}^+\}$$

is not defined. Let us include a supremum $\top$ denoted ∞ to $\mathbb{R}^+$, that is by definition :

$$\infty = \underline{MAX}\{i \mid i \in \mathbb{R}^+\}$$

Now $\mathbb{R}^+$ ($\leq$, 0, ∞, max, min, MAX, MIN) is a complete lattice (Birkhoff [1973]).

3.3 *Solving the Equations*

Let us simplify the equations by elimination of the variables C_1 and C_3 (and supposing that the value of $p(\alpha)$ is given by p) :

$$\begin{cases} C_2 = 1 + C_2 \cdot p \\ C_4 = C_2 \cdot (1 - p) \end{cases}$$

C_2 depends only on itself so that we can solve first the subsystem :

$$\{C_2 = 1 + C_2 \cdot p$$

i.e. $C_2 = F(C_2)$ where $F(x) = 1 + x \cdot p$.

It is obvious that the solutions of this equation are $1/(1-p)$ and ∞. However let us go on with the example applying the theorems of paragraphs 3.3 and 3.4.

The function F is order-preserving, since :

$$\{x \leq y\} \Longrightarrow \{1 + x \cdot p \leq 1 + y \cdot p\}$$

(Recall $0 \leq p \leq 1$).

Note

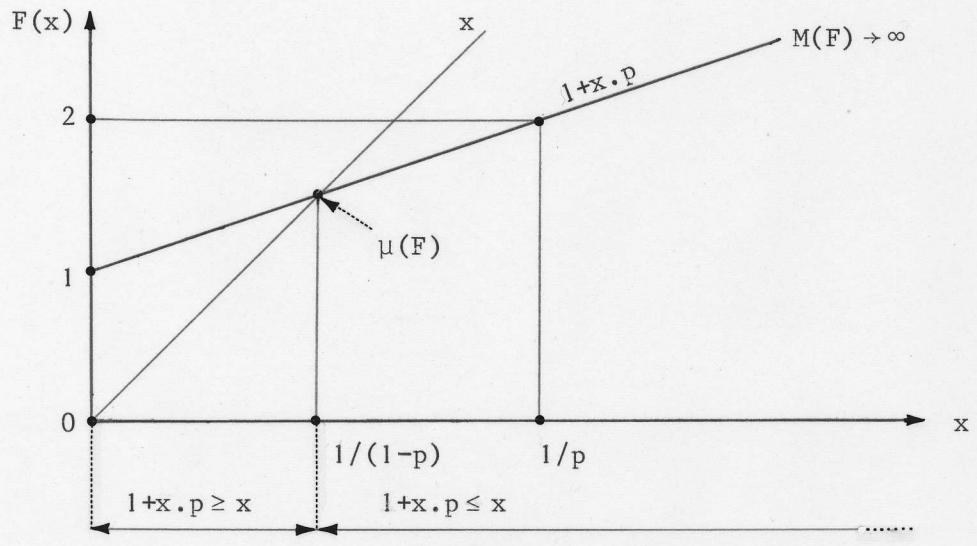
The prove that F is order-preserving needs not be done for every particular program. In general it is possible to show that the isotony (or continuity) of F is a direct consequence of the syntactic method which is utilized to built the system of equations.

Tarski's theorem then states that the extreme fixpoints of F are :

$$\begin{aligned} \mu(F) &= \underline{MIN}\{x \in \mathbb{R}^+ \mid 1 + x.p \leq x\} \\ &= \underline{MIN}[1 / (1-p), \infty] \\ &= 1 / (1-p) \end{aligned}$$

$$\begin{aligned} M(F) &= \underline{MAX}\{x \in \mathbb{R}^+ \mid 1 + x.p \geq x\} \\ &= \underline{MAX}\{[0, 1 / (1-p)] \cup \{\infty\}\} \\ &= \infty \end{aligned}$$

which is easily understood by the following geometric interpretation :



Note that this definition of the fixpoint is not constructive, however the approximation sequences give an algorithm to compute them.

The map F is clearly continuous since it is infinitely distributive that is for any 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 fixpoint :

$$\lambda_0 = \infty$$

$$\lambda_1 = F(\lambda_0) = 1 + \infty \ . \ p = \infty$$

$$= \lambda_0$$

— The ascending approximation sequence leads to the minimal fixpoint :

$$\lambda_0 = 0$$

$$\lambda_1 = F(\lambda_0) = 1 + 0 \ . \ p = 1$$

$$\lambda_2 = F(\lambda_1) = 1 + 1 \ . \ p$$

$$\lambda_3 = F(\lambda_2) = 1 + (1 + p) \ . \ p = 1 + p + p^2$$

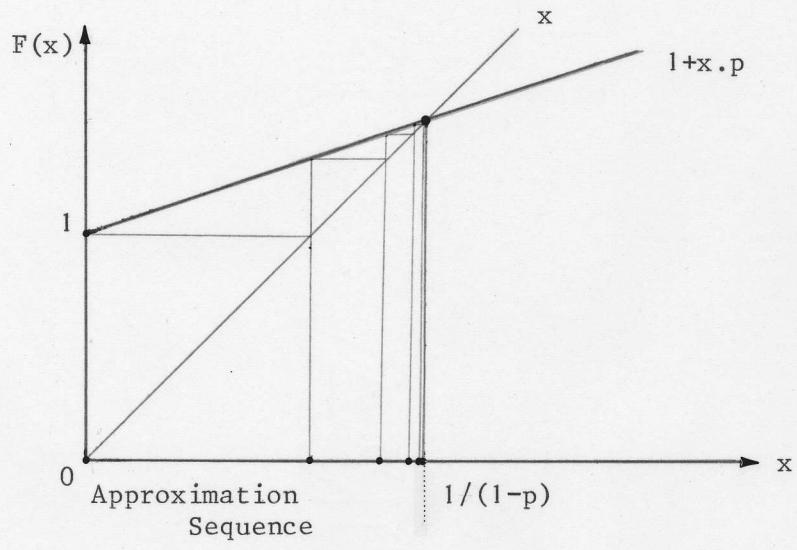
...

$$\lambda_n = F(\lambda_{n-1}) = 1 + p + p^2 + \ ... \ + p^{n-1}$$

...

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

— The classical geometric interpretation is the following :



Notice that this abstract interpretation of programs leads to a system of linear equations. The increasing approximation sequence corresponds to the Jacobi's iterative method known as "successive approximations" (for numerical coefficients).

4. COPING WITH INFINITE APPROXIMATION SEQUENCES

Since approximation sequences may be infinite the algorithmic construction of exact solutions as their limits may be impossible. Hence other methods must be used to analyze the program properties. We can roughly classify these methods as follows :

4.1 Verification of the Correctness of a Solution Provided by the Programmer

4.1.1 A solution S of the system of equations is provided by the user. The problem simply consists in verifying that $F(S) = S$.

4.1.2 The user may also provide the solution of the equations by giving the value $S(n)$ of the general term S_n of the approximation sequence $S_0, \ldots, S_n, \ldots$. The problem is then to verify that $S(0) = \bot^k$ (or $\top^k$) and $S(n+1) = F(S(n))$. The solution S of the equations is obtained by passing to the limit $S = \lim_{n\to\infty} (S(n))$.

4.2 Automatized Computation of the Exact Solution

4.2.1 The exact solution of the equations may be obtained by the limit of the approximation sequence when the length of this sequence is finite.

For infinite approximation sequences the exact solution may also be obtained by other resolution methods :

4.2.2 Formal resolution of the equations by eliminations and simplifications,

4.2.3 Resolution of difference equations as follows : the recurrence relationship $S_0 = \perp^k$ and $S_{n+1} = F(S_n)$ between consecutive terms of the approximation sequence leads to a system of difference equations, which may be solved to get S_n as a function of n, $S_n = S(n)$ which gives the least solution S as $\lim_{n\to\infty}(S(n))$.

4.3 Verification of Properties of the Exact Solutions

Generally some properties of programs may be proved to hold without full knowledge of the solution S of the system of equations. It just suffices to prove some property P(S) of S. Since the solution S of the equations is defined as the limit of an approximation sequence S_0, ..., $S_n = F^n(S_0)$, ... one can prove P(S) using one of the following induction rules :

4.3.1 From $\{P(S_0)$ and $\{P(F^n(S_0)) \Longrightarrow P(F^{n+1}(S_0))\}\}$ infer P(S).
(e.g. De Bakker and Scott[1969]).

4.3.2 From $\forall i\ \{(\forall j \mid j < i)\ P(F^i(S_0)) \Longrightarrow P(F^{i+1}(S_0))\}$ infer P(S).
(e.g. Morris[1971]).

The previous rules may be also written more stylishly :

4.3.3 From $\{P(S_0)$ and $\{\{(\forall X)P(X)\} \Longrightarrow \{P(F(X))\}\}\}$ infer P(S).

Warning : These induction rules may be used only for "admissible" predicates P. (See e.g. Manna[1974]).

4.4 Approximation of the Solution

When the approximation sequences are infinite one can approximate the least (greatest) fixpoint S of F by an upper approximation S_u such that $S \sqsubseteq_k S_u$ and S_u is "sufficiently close" to S (or by a lower approximation S_ℓ such that $S_\ell \sqsubseteq_k S$ and S_ℓ is "sufficiently close" to S).

The approximates S_ℓ and S_u may be obtained by strengthening the iterates in the increasing or decreasing approximation sequences (e.g. Cousot[1976], Cousot[1977a], Sintzoff[1976]).

Note that in methods 4.1, 4.2, 4.3 one can also use approximations of the solution (provided that one can define what is a correct approximation of the solution , this is usually the purpose of the partial ordering $\sqsubseteq_k$).

4.5 *Example : Performance Analysis of Programs*

Numerous techniques for static analysis of programs use the above alternative methods and this will be examplified in the subsequent paragraphs.

Let us see for example how 4.2.3 and 4.2.2 permit to understand the papers of Wegbreit[1975a] and Kennedy and Zucconi[1977] dealing with performance analysis of programs.

The linear equations may be solved by establishing a recurrence relationship between consecutive terms of the ascending approximation sequence. This leads to a system of difference equations which may be automatically solved (Cohen and Katcoff[1976]) :

Example

The recursive equation $C_2 = 1 + C_2.p$ has a solution $C_2(\infty)$ defined by :

$$\begin{cases} C_2(0) = 0 \\ C_2(n+1) = 1 + C_2(n) \cdot p \end{cases}$$

These simple difference equations have the solution :

$$C_2(n) = (1 + p^n)/(1 - p)$$

and $\lim_{n\to\infty} p^n = 0$ since $0 \leq p < 1$ thus $C(\infty) = 1/(1 - p)$, (or ∞ if $p = 1$).

The linear equations may also be solved by formal substitutions, successively applying simplification rules until obtaining equations which solution is known. This is the method usually used when solving equations by hand.

The approximates S_ℓ and S_u may be obtained by strengthening the iterates in the increasing or decreasing approximation sequences (e.g. Cousot[1976], Cousot[1977a], Sintzoff[1976]).

Note that in methods 4.1, 4.2, 4.3 one can also use approximations of the solution (provided that one can define what is a correct approximation of the solution , this is usually the purpose of the partial ordering $\sqsubseteq_k$).

4.5 *Example : Performance Analysis of Programs*

Numerous techniques for static analysis of programs use the above alternative methods and this will be examplified in the subsequent paragraphs.

Let us see for example how 4.2.3 and 4.2.2 permit to understand the papers of Wegbreit[1975a] and Kennedy and Zucconi[1977] dealing with performance analysis of programs.

The linear equations may be solved by establishing a recurrence relationship between consecutive terms of the ascending approximation sequence. This leads to a system of difference equations which may be automatically solved (Cohen and Katcoff[1976]) :

Example

The recursive equation $C_2 = 1 + C_2.p$ has a solution $C_2(\infty)$ defined by :

$$\begin{cases} C_2(0) = 0 \\ C_2(n+1) = 1 + C_2(n) \cdot p \end{cases}$$

These simple difference equations have the solution :

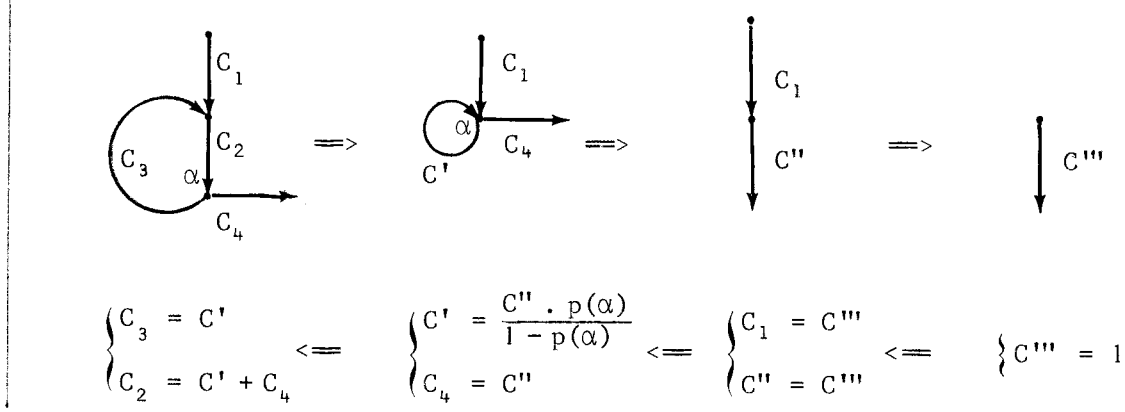
$$C_2(n) = (1 + p^n)/(1 - p)$$

and $\lim_{n\to\infty} p^n = 0$ since $0 \leq p < 1$ thus $C(\infty) = 1/(1 - p)$, (or ∞ if $p = 1$).

The linear equations may also be solved by formal substitutions, successively applying simplification rules until obtaining equations which solution is known. This is the method usually used when solving equations by hand.

This simplification process is generally presented as a sequence of reductions of the program graph by clementary transformations (Graham[1972], Kennedy and Zucconi[1977]).

Example



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", which is a frequent case but not a general one (Kennedy and Zucconi[1977]).

5. APPLICATION TO GLOBAL DATA FLOW ANALYSIS

In analyzing a computer program for purposes of code improvement, it is necessary to be able to trace at compile time the flow of information through a program. This analysis is called "global data flow analysis", and is used in connection with code improvement techniques (such as constant propagation, common subexpression elimination, moving invariant computations out of loops or reduction of the number of store-load sequences between memory and high-speed registers), (Aho and Ullman[1973], Allen[1971], Branquart et al.[1973], Cocke[1970], Fong et al.[1975], Hecht[1975], Hecht and Ullman[1973], Jensen[1965], Kam and Ullman[1976], Kennedy[1971], Kildall[1973], Morel and Renvoise[1974], Schaefer[1973], Schwartz[1975], Ullman[1975], Urschler[1974], Wegbreit[1975b]).

Global data flow analysis involves solving a class of problems each of which can be dealt with essentially the same manner that is solving a system of equations established by a suitable interpretation of the program.

The classical systematic techniques for global flow analysis, that is :

(a) - The Cocke-Allen interval analysis, typified by Allen[1971] and Cocke[1970].

(b) - The iterative methods, typified by Hecht and Ullman[1973].

differ only by the way they solve the system of equations.

Methods (a) are a graph formulation of an algorithm which formally solves the equations (see 4.2.2). They are applicable only to a limited class of recursive equations (corresponding to "reducible" program flow graphs) and to a limited class of interpretations.

Methods (b) correspond to the resolution of the equations by successive approximations (see 4.2.1) and therefore are not subject to the limitations of the Cocke-Allen interval approach.

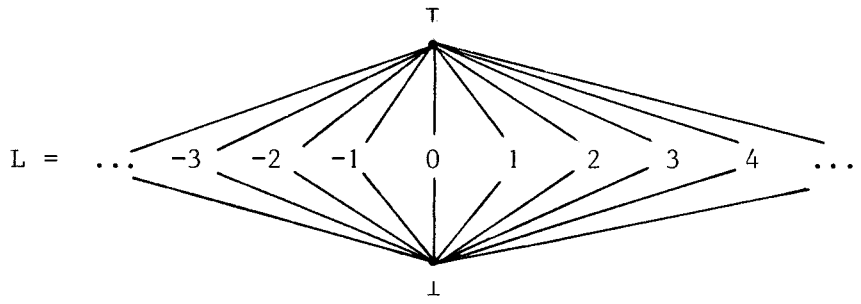
(Aside, given a particular interpretation the practical question of which of the two approaches is the most efficient has not yet received a conceptual answer, see a.o. Kennedy[1976]).

5.1 Constant Propagation

A fairly simple case of program analysis and optimization occurs when constant computations are evaluated at compile-time. Consider the following skeletal program, Kildall[1973] :

```
{1}   a := 1; c := 0;
{2}   while ... do
{3}         b := 2;
{4}         d := a+b;
{5}         e := b+c;
{6}         c := 4;
{7}   od;
```

The abstract contexts associated to the various program points are state vectors P_1, ..., P_7 which associate an abstract value with each of the variables of the program. We will note $P_3(b)$ the abstract value of b in the abstract context associated to program point {3}. The set of abstract values is the infinite lattice :



The symbol $\top$ will be the value of non-constant variables.

If the program has n variables x_1, ..., x_n then abstract contexts are state vectors belonging to the product lattice L^n.

The system of equations corresponding to our example is :

$$\begin{cases} P_1 = \langle \top, \top, \top, \top, \top \rangle \\ P_2 = P_1(a \leftarrow 1)(c \leftarrow 0) \\ P_3 = P_2 \sqcup_5 P_7 \\ P_4 = P_3(b \leftarrow 2) \\ P_5 = P_4(d \leftarrow P_4(a) \boxplus P_4(b)) \\ P_6 = P_5(e \leftarrow P_5(b) \boxplus P_5(c)) \\ P_7 = P_6(c \leftarrow 4) \end{cases}$$

with the following notations :

- $P(x \leftarrow v)$ is the state vector P where the value of variable x is changed to v, hence : $P(x \leftarrow v)(y) =$ *if* x = y *then* v *else* P(y) *fi*
- $P \sqcup_5 Q$ designates the component by component union of the lattice L^5. Therefore $(P \sqcup_5 Q)(x) = P(x) \sqcup Q(x)$ for x = a, b, ..., e where $\sqcup$ is defined by :

$$\top \sqcup x = x \sqcup \top = \top, \ \forall x \in L$$
$$\bot \sqcup x = x \sqcup \bot = x, \ \forall x \in L$$
$$i \sqcup j = \text{if } i = j \text{ then } i \text{ else } \top \text{ fi}, \ \forall i,j \in \mathbb{N}$$

- $\oplus$ is the abstract addition operator :

$$\bot \oplus x = x \oplus \bot = \bot, \ \forall x \in L$$
$$\top \oplus x = x \oplus \top = \top, \ \forall x \in L - \{\bot\}$$
$$i \oplus j = i + j, \ \forall i,j \in \mathbb{N}$$

Notice that the system of equations is of the form :

$$X = F(X) \quad \text{with} \quad F : (L^5)^7 \to (L^5)^7$$

Since every strictly increasing chain of L is of length at most 3, $(L^5)^7$ satisfies the ascending chain condition (see 2.4.1). This implies that the iterative method of resolution of the system of equations converges in at most $3 \times 5 \times 7 + 1$ successive iterations.

The successive approximations are the following :

	a	b	c	d	e	
$P_i \quad 1 \leq i \leq 7$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	Initialisation
P_1	$\top$	$\top$	$\top$	$\top$	$\top$	
P_2	1	$\top$	0	$\top$	$\top$	
P_3	1	$\top$	0	$\top$	$\top$	
P_4	1	2	0	$\top$	$\top$	Step 1
P_5	1	2	0	3	$\top$	
P_6	1	2	0	3	2	
P_7	1	2	4	3	2	
P_3	1	$\top$	$\top$	$\top$	$\top$	
P_4	1	2	$\top$	$\top$	$\top$	
P_5	1	2	$\top$	3	$\top$	Step 2
P_6	1	2	$\top$	3	$\top$	
P_7	1	2	4	3	$\top$	
P_3	1	$\top$	$\top$	$\top$	$\top$	
P_4	1	2	$\top$	$\top$	$\top$	
P_5	1	2	$\top$	3	$\top$	Step 3, stabilization, stop
P_6	1	2	$\top$	3	$\top$	
P_7	1	2	4	3	$\top$	

The final result is that "a" is constant equal to 1, and "b" and "d" are constant in the loop as soon as they have been assigned.

5.2 *Live Variables*

This example is representative of the "boolean techniques of program optimization". It has been intensively studied in the litterature (a.o., Allen and Cocke[1976], Hecht and Ullman[1973], Kennedy[1971], Schaefer[1973], Ullman[1973]) so that very efficient algorithms exist (Aho and Ullman[1975], Kennedy[1975], Tarjan[1976]).

Given a variable X, which is defined at various points in a program, we wish to determine for each point p in a program flow graph whether or not X will be used after control leaves p. We say that X is *live* at p if it can be used again and *dead* at p otherwise. The "live" information would be useful in register allocation for example since the value of a variable which can never be used again needs not be saved.

Suppose the program has been represented by its control flow graph in which each node represents a basic block and each edge represents a possible block to block transfer.

For each block b in the program we have to determine the set live(b) of variables X for which there is a path from the entry point of b to a use of X, which path is *definition-clear with respect to* X (i.e. contains no redefinition of the variable X).

Let use(b) be the set of variables which have exposed uses in block b, i.e., those variables with a definition-clear path from the entry of block b to a use within b.

Let clear(b) be the set of variables X for which the path through the basic bloc b is definition-clear with respect to X.

Note that the sets used(b) and clear(b) can be computed by a local examination of block b.

Now there exists an X-definition-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.
In equation form :

$$\underline{live}(b) = \underline{use}(b) \cup \left(\bigcup_{x \in Succ(b)} (\underline{clear}(b) \cap \underline{live}(x)) \right) \qquad \underline{fig}.\ 5.2.a$$

For an exit node which has no successors and contains no commands we have :

$$\underline{live}(e) = \emptyset$$

Note

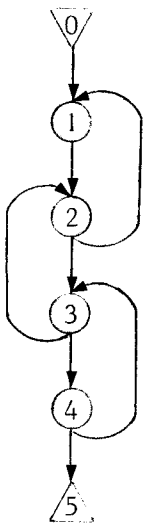
Since the above system of equations does not necessarily have a unique solution we want the smallest such solution (the ordering is set inclusion $\subseteq$).

Consider for example a program such as :

begin i := 1 ; while ... do ... od ; j := i end.

where "i" and "j" are not used in the loop body. The greatest solution would consider "i" and "j" to be live in the loop body, whereas the smallest solution will only consider "i" to be live in the loop).

Let us illustrate the determination of the live sets for a simple flow graph with two variables α and β :



$\underline{use}(0) = \emptyset$; $\underline{clear}(0) = \emptyset$

$\underline{use}(1) = \{\beta\}$; $\underline{clear}(1) = \{\alpha, \beta\}$

$\underline{use}(2) = \emptyset$; $\underline{clear}(2) = \{\alpha, \beta\}$

$\underline{use}(3) = \emptyset$; $\underline{clear}(3) = \{\alpha, \beta\}$

$\underline{use}(4) = \{\alpha\}$; $\underline{clear}(4) = \{\alpha\}$

$\underline{live}(5) = \emptyset$

Thus the variable α is defined in node 0 and used in node 4, whereas variable β is defined in nodes 0 and 4 and used in node 1.

The ascending approximation sequence leads to the following trace :

live(b) / Steps \ b	0	1	2	3	4	5
Initialization	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
1	$\emptyset$	$\{\beta\}$	$\{\beta\}$	$\{\beta\}$	$\{\alpha\}$	-
2	$\emptyset$	$\{\beta\}$	$\{\beta\}$	$\{\alpha, \beta\}$	$\{\alpha\}$	-
3	$\emptyset$	$\{\beta\}$	$\{\alpha, \beta\}$	$\{\alpha, \beta\{$	$\{\alpha\}$	-
4	$\emptyset$	$\{\alpha, \beta\}$	$\{\alpha, \beta\}$	$\{\alpha, \beta\}$	$\{\alpha\}$	-
5	$\emptyset$	$\{\alpha, \beta\}$	$\{\alpha, \beta\}$	$\{\alpha, \beta\}$	$\{\alpha\}$	-

And therefore 20 applications of equation 5.2.a are needed, plus 5 more applications which are required before no change to the sets is recognized which then terminates the iteration process.

5.3 *Optimal Approximation Sequence*

Notice that the fixpoint equation :

$$X = F(X)$$

should be solved, according to the definition of the approximation sequence (2.4.1) by Jacobi's method of successive approximations :

$$X^{r+1} = F(X^r) \quad (r = 0, 1, 2, \ldots)$$

which can be detailled in :

$$\begin{cases} x_i^{r+1} = f_i(x_1^r, x_2^r, \ldots, x_k^r) \\ i = 1, 2, \ldots, k \end{cases} \qquad (r = 0, 1, 2, \ldots)$$

In fact, we used the Gauss-Seidel method on F :

$$\begin{cases} x_1^{r+1} = f_1(x_1^r, x_2^r \ldots\ldots\ldots\ldots x_k^r) \\ \vdots \\ x_i^{r+1} = f_i(x_1^{r+1} \ldots x_{i-1}^{r+1}, x_i^r \ldots x_k^r) \\ \vdots \\ x_k^{r+1} = f_k(x_1^{r+1}, \ldots\ldots\ldots x_{k-1}^{r+1}, x_k^r) \end{cases}$$

which consists in continually reinjecting in the computations the last results of the computations themselves. This reduces the memory congestion and accelerates the convergence. However Robert[1976] shows that Gauss-Seidel method is not algorithmicly more reliable than Jacobi's successive approximation method. Without sufficient hypothesis on F Jacobi's method may converge although the one of Gauss-Seidel cycles. The contrary is also true, that is Gauss-Seidel's method may converge although Jacobi's iterations diverge.

Fortunately the continuity hypothesis on F implies that any chaotic iteration method converges to the right solution. Otherwise stated this signifies that one can randomly determine at each step which are the components of the system of equations which will evolve (under the condition to never forget one definitively).

Although the convergence to the least solution does not depend on the order of computations the efficiency of the iteration process does. Until now the interesting question of determining the optimal approximation sequence has received no general conceptual answer and is unlikely to receive one (if we think to numerical analysis where the problem has been extensively studied using very peculiar hypothesis).

However, experimental and theoretical results have been obtained for particular applications. With regard to the problem of live variables the optimal approximation sequence has been shown to exist ("node listing" method of Kennedy[1975], generalized by Tarjan[1976]), and may be algorithmicly specified for particular abstract interpretations (on reducible

flow graphs, Aho and Ullman[1975], although these results are expressed in terms of order of search over a flow graph).

Example

Initializing live(b) by used(b) and applying the equation live(b) in the order b = 1, 2, 3, 4, 3, 2, 1, 0 ; we have :

live(b) Steps \ b	0	1	2	3	4	5
Initialization	$\emptyset$	$\{\beta\}$	$\emptyset$	$\emptyset$	$\{\alpha\}$	$\emptyset$
1		$\{\beta\}$				
2			$\{\beta\}$			
3				$\{\beta\}$		
4					$\{\alpha\}$	
5				$\{\alpha, \beta\}$		
6			$\{\alpha, \beta\}$			
7		$\{\alpha, \beta\}$				
8	$\emptyset$					

Therefore 8 applications of the equation 5.2.a are strictly necessary (instead of 20) and no supplementary applications are needed to prove that the iteration process has converged (instead of 5).

6. APPLICATION TO LOGICAL ANALYSIS OF PROGRAMS

The method introduced by Floyd[1967] and Naur[1966] for proving correctness of programs has been intensively studied, extended and even partly automatized. Its presentation as an abstract interpretation of programs

permits to understand the effective originality and respective power of various methods which are often stated to be new and different from previous ones. Moreover, this permits to elucidate several mis-understood problems.

6.1 *Abstract Contexts*

In this interpretation abstract contexts 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 ($\sqsubseteq$, $\bot$, $\top$, $\sqcup$, $\sqcap$, $\bigsqcup$, $\bigsqcap$) by choosing respectively (==>, false, true, or, and, OR, AND).

6.2 *System of Logical Forward Equations*

We use the notation "$\{P(X, \overline{X})\}$ 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 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 (Dijkstra[1976], Hoare[1969]) the following rules permit to associate a system of equations with any flowchart program :

Program entry point :

$$\{(x_i = v_i),\ i = 1,\ \ldots,\ m\}$$

The respective initial values of the variable $x_1, \ldots, x_m$ are the symbols $v_1, \ldots, v_m$. The v_i may eventually be Ω denoting the uninitialized value.

Assignment statements :

$$\{P\} \quad i := E \quad \{\exists i' \mid P(i \leftarrow i') \text{ and } (i = E(i \leftarrow i'))\}$$

We note "$\alpha(x \leftarrow y)$" a 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 :

$$\{P\} \quad \underline{\text{if}}\ Q\ \underline{\text{then}}\ \{P\ \underline{\text{and}}\ Q\} \quad \langle\text{statement list}\rangle \quad \underline{\text{fi}}$$

Go to statements and labels :

$$L : \{ \underset{i \in \underline{\text{pred}}(L)}{\underline{\text{OR}}} P_i \}$$

where $\underline{\text{pred}}(L)$ denotes the set of program points going to L (sequentially or by an unconditional jump "$\underline{\text{go to}}$ L" to the constant label L).

We will examplify the above rules on the very simple program (over the integers $\mathbb{N}^* = \mathbb{N} \cup \{\Omega\}$) :

```
      {P0}
              i := 1;
      {P1}
loop: {P2}
              if i ≤ 1000 then
      {P3}
                   i := i+1 ;
      {P4}
                   go to loop ;
              fi ;
      {P5}
```

fig 6.2.a

The corresponding system of forward equations is the following :

$$\begin{cases} (1) & P_0 = (i = \Omega) \\ (2) & P_1 = \{\exists i' \in \mathbb{N}^* \mid P_0(i \leftarrow i') \text{ and } (i = 1)\} \\ (3) & P_2 = P_1 \text{ or } P_4 \\ (4) & P_3 = P_2 \text{ and } (i \leq 1000) \\ (5) & P_4 = \{\exists i' \in \mathbb{N}^* \mid P_3(i \leftarrow i') \text{ and } (i = i'+1)\} \\ (6) & P_5 = P_2 \text{ and } (\text{not}(i \leq 1000)) \end{cases}$$

fig 6.2.b

6.3 Optimal Invariants

The system of equations has in general several solutions.

Example

Let us prove that the assertion :

$$P_2 = (i \leq 1001)$$

is an admissible invariant at program point {2}.

Replacing in the system of equations we get :

$$P_0 = (i = \Omega)$$

$$P_1 = \{\exists i' \in \mathbb{N}^* \mid P_0(i \leftarrow i') \text{ and } (i = 1)\}$$
$$= \{\exists i' \in \mathbb{N}^* \mid (i' = \Omega) \text{ and } (i = 1)\}$$
$$= (i = 1)$$ (since i' is not a program variable and therefore is uninitialized).

$$P_3 = P_2 \text{ and } (i \leq 1000)$$
$$= (i \leq 1001) \text{ and } (i \leq 1000)$$
$$= (i \leq 1000)$$

$$P_4 = \{\exists i' \in \mathbb{N}^* \mid P_3(i \leftarrow i') \text{ and } (i = i'+1)\}$$
$$= \{\exists i' \in \mathbb{N}^* \mid (i' \leq 1000) \text{ and } (i = i'+1)\}$$
$$= (i \leq 1001)$$

$$P_5 = P_2 \text{ and not } (i \leq 1000)$$
$$= (i \leq 1001) \text{ and not } (i \leq 1000)$$
$$= (i = 1001)$$

It is now very easy to verify that the inductive invariant P_2 is correct since :

$$\begin{aligned} P_2 &= P_1 \underline{\text{ or }} P_4 \\ &= (i = 1) \underline{\text{ or }} (i \leq 1001) \\ &= (i \leq 1001) \end{aligned}$$

The same way one can prove that $P_2 = \{1 \leq i \leq 1001\}$ is another admissible solution at program point {2}.

Yet according to Tarski's theorem (see 2.3) there exists a least solution S_{opt} (least for ordering $\sqsubseteq_5$ that is $\Longrightarrow_5$). We call this least solution the *optimal invariants* since they imply any other solution of the system of equations.

Proof

Since S_{opt} is the infimum in the complete lattice of fixpoints of F (Tarski's theorem 2.3) we have :

$$(\forall P \mid P = F(P)),\ S_{opt} \sqsubseteq P$$

$$(\forall P \mid P = F(P)),\ S_{opt} \Longrightarrow P$$

Example

The optimal invariants for program of <u>fig</u> 6.2.a are the least solution of the system of equations of <u>fig</u> 6.2.b that is :

$$\begin{aligned} P_0 &= (i = \Omega) \\ P_1 &= (i = 1) \\ P_2 &= (1 \leq i \leq 1001) \\ P_3 &= (1 \leq i \leq 1000) \\ P_4 &= (2 \leq i \leq 1001) \\ P_5 &= (i = 1001) \end{aligned}$$

6.4 <u>Proof of Total Correctness</u>

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 S_{opt}.
- 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 :

  $$S_{opt}(h)(\overline{Y}, \overline{X}) \text{ and } (S_{opt}(h)(\overline{Y}, \overline{X}) \Rightarrow \psi(\overline{Y}, \overline{X}))$$

  which requires that the program terminates at some haltpoint h with a state $\overline{Y}$ of the program variables satisfying the output specification ψ.

In formulas (adapted from Katz and Manna[1976]) we must prove :

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

Where, from what preceded we have :

$$S_{opt} = \mu(F).$$

Example

Consider the program segment :

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

fig 6.4.a

Applying Hoare's rules we obtain the following system of equations :

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

fig 6.4.b

and the corresponding optimal invariants are :

$$P_1 = (x = x_0) \text{ and } (y = y_0)$$

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

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

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

fig 6.4.c

Suppose now that one wants to prove that for any input values (x_0, y_0) of (x, y) satisfying the input specification :

$$\Phi(x_0, y_0) = \{(x_0 \geq 0) \text{ and } (y_0 \geq 0)\}$$

the program terminates with output specification :

$$\psi(x, y, x_0, y_0) = \{(x \geq 0) \text{ and } (y \geq 0) \text{ and } (x < y)\}$$

We must prove

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

After trivial simplifications this consists in proving that :

$$(x_0 \geq 0) \text{ and } (y_0 \geq 0) \text{ implies}$$

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

We know from arithmetics 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 ($x_0 < 0$ and $x_0 \geq 0$). For the same reason $P_4(x, y, x_0, 0)$ is always false when $x_0 \geq 0$ hence the program must enter an infinite loop for such input values.

6.5 *Approximate Invariants, Systems of Inequations and Proofs of Partial Correctness*

Most program proving methods use inequations of the form :

$$P_i \mathrel{<=} f_i(P_1, \ldots, P_k) \qquad (i = 1, \ldots, k)$$

whereas we used the equations :

$$P_i = f_i(P_1, \ldots, P_k) \qquad (i = 1, \ldots, k)$$

For example, instead of :

$$\{i > 0\} \quad i := i+1 \quad \{i > 1\}$$

one can legally write a less precise assertion, such as :

$$\{i > 0\} \quad i := i+1 \quad \{i > 1/2\}$$

since the strongest post-condition resulting from the pre-condition $\{i > 0\}$ is $\{i > 1\}$ which implies $\{i > 1/2\}$.

Tarski's theorem (2.3) implies that :

$$S_{opt} = \mu(F) = \Pi(P \mid F(P) \subseteq P)$$

$$\Longrightarrow \quad \forall P,\ (F(P) \subseteq P) \Longrightarrow (S_{opt} \subseteq P)$$

$$\Longrightarrow \quad \forall P,\ (F(P) \Longrightarrow P) \Longrightarrow (S_{opt} \Longrightarrow P)$$

Therefore in order to find a correct *approximate invariant* S hence such that :

$$S_{opt} \Longrightarrow P$$

it suffices to find a solution S to the system of inequations

$$P \mathrel{<=} F(P)$$

According to Katz and Manna[1976] the reasoning for proving the partial correctness of a program π is the following :

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

$$(\exists P \mid P <= F(P)),\ (\forall \overline{X} \mid \Phi(\overline{X})),\ \forall h,\ \forall \overline{Y}$$
$$P(h)(\overline{Y}, \overline{X}) => \psi(\overline{Y}, \overline{X})$$

We can easily show that this reasoning is correct since under the hypothesis that π terminates we have :

$$\exists h,\ \exists \overline{Y} \mid S_{opt}(h)(\overline{Y}, \overline{X})$$

and if the program is partially correct we have

$$P(h)(\overline{Y}, \overline{X}) => \psi(\overline{Y}, \overline{X})$$

but since $P <= F(P)$ we have $S_{opt} => P$ hence

$$S_{opt}(h)(\overline{Y}, \overline{X}) => \psi(\overline{Y}, \overline{X})$$

Example

Suppose we want to prove the partial correctness of the simple program of fig 6.2.a, for an input specification $\Phi(i, i_0) =$ true and an output specification $\psi(i, i_0) = (i = 1001)$. The system of inequations corresponding to that program is easily derived from the equations 6.2.b. This system of inequations admits an infinity of solutions. For example, we can easily derive an infinity of approximation invariants from the inductive invariants :

$$P_2 = (a \leq i \leq 1001) \text{ for any } a \leq 1$$

replacing in the system of equations we get :

$$P_0 = (i = \Omega)$$
$$P_1 = (i = 1)$$
$$P_3 = P_2 \text{ and } (i \leq 1000)$$
$$= (a \leq i \leq 1000)$$
$$P_4 = \{\exists i' \in \mathbb{N}^* \mid P_3(i \leftarrow i') \text{ and } (i = i'+1)\}$$
$$= \{\exists i' \mid (a \leq i' \leq 1000) \text{ and } (i = i'+1)\}$$
$$= ((a+1) \leq i \leq 1001)$$

It is now very easy to verify that the approximate inductive invariant is correct since :

$P_2 <= P_1$ or P_4

$P_2 <= (i = 1)$ or $((a+1) \leq i \leq 1001)$

which is obvious since $P_2 = (a \leq i \leq 1001)$ and $a \leq 1$.

Now the output approximate invariant is :

$P_5 = P_2$ and not $(i \leq 1001)$

$= (a \leq i \leq 1001)$ and $(1000 < i)$

$= (i = 1001)$

and obviously $P_5 \Rightarrow \psi$.

Discussion

The verification of program partial correctness has been shown to be amenable to mechanization (see a.o. Deutsch[1973], King[1969], Waldinger and Levitt[1974]). 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 of the inequations since a simple propagation in the equations permits to deduce the remaining invariants. Thus program partial correctness methods consists in verifying that a solution of the inequations provided by the user is correct (see 4.1.1). and that this solution implies the output specification when assuming the input specifications to hold.

The only criterion to choose the inductive approximate invariants among the infinitely many ones is that they must be simple enough to be easily proved by the theorem prover and yet powerful enough for the theorem prover to deduce the output specification. This criterion is certainly not a useful guide for the user to discover the convenient inductive invariants.

6.6 Approximate Invariants, Systems of Inequations and Proofs of Termination

Let π be a program which 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 :

$$\{(\forall \overline{X} \mid \Phi(\overline{X})),\ \exists h,\ \exists \overline{Y} \mid S_{opt}(h)(\overline{Y},\ \overline{X})\}$$

where $S_{opt} = \mu(F)$.

Applying Tarski's theorem (see 2.3) we have :

$$S_{opt} = \underline{AND}\ \{P \mid F(P) \Rightarrow P\}$$

the termination condition becomes :

$$\{(\forall \overline{X} \mid \Phi(\overline{X})),\ \exists h,\ \exists \overline{Y} \mid \underline{AND}\ \{P(h)(\overline{Y},\ \overline{X}) \mid F(P) \Rightarrow P\}\}$$
$$= \{(\forall P \mid P <= F(P)),\ (\forall \overline{X} \mid \Phi(\overline{X})), \exists\ h,\ \exists \overline{Y} \mid P(h)(\overline{Y},\ \overline{X})\}$$

which is the termination condition of Katz and Manna[1976].

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 inequations $P <= F(P)$ which admits infinitely many solutions. This fact is not surprising at all since this condition is based on Tarski's theorem which is not constructive. Their repartee was to use other methods to prove termination such as Floyd [1967]'s method based on well-founded sets with no infinitely descending chains.

Yet our termination condition is utilizable provided that we may find a constructive definition of the optimal invariants S_{opt}. The ascending sequence of successive approximations is the key to that problem.

6.7 Discovery of Optimal Invariants

The optimal invariants S_{opt} are the limit of the ascending approximation sequence starting from the infimum false. Let us use the system of equations of fig 6.2.b as example :

Initialization :

$$P_i = \underline{false} \qquad i = 0..5$$

Iteration 1 :

$P_0 = (i = \Omega)$

$P_1 = \{\exists i' \in \mathbb{N} \mid P_0(i \leftarrow i') \text{ and } (i = 1)\}$

$= \{\exists i' \in \mathbb{N} \mid (i' = \Omega) \text{ and } (i = 1)\}$

$= (i = 1)$

$P_2 = P_1 \text{ or } P_4$

$= (i = 1) \text{ or false}$

$= (i = 1)$

$P_3 = P_2 \text{ and } (i \leq 1000)$

$= (i = 1) \text{ and } (i \leq 1000)$

$= (i = 1)$

$P_4 = \{\exists i' \in \mathbb{N} \mid P_3(i \leftarrow i') \text{ and } (i = i'+1)\}$

$= (i = 2)$

$P_5 = P_2 \text{ and } (i > 1000)$

$= (i = 1) \text{ and } (i > 1000)$

$= \text{false}$

Iteration 2 :

$P_0 = (i = \Omega)$

$P_1 = (i = 1)$

$P_2 = P_1 \text{ or } P_4$

$= (i = 1) \text{ or } (i = 2)$

$= (1 \leq i \leq 2)$

$P_3 = (1 \leq i \leq 2)$

$P_4 = (2 \leq i \leq 3)$

$P_5 = \text{false}$

...

Iteration k :

$P_0 = (i = \Omega)$

$P_1 = (i = 1)$

$P_2 = (1 \leq i \leq k)$

$P_3 = (1 \leq i \leq k)$

$P_4 = (2 \leq i \leq k+1)$

$P_5 = \text{false}$

...

Iteration 1001 :

$P_0 = (i = \Omega)$

$P_1 = (i = 1)$

$P_2 = (1 \leq i \leq 1001)$

$P_3 = (1 \leq i \leq 1000)$

$P_4 = (2 \leq i \leq 1001)$

$P_5 = (i = 1001)$

At iteration 1002 we find the same result, so that further iterations are useless, the iterates have converged. In 1001 + 1 iterations, the optimal invariants are generated directly from the system of equations.

It is a bit astonishing to discover that the approximation sequence really describes the execution of the program. Yet in general it is a means of computation which differs from usual execution since all possible program paths are followed parallelly, and the paths are initiated from all program points (and not from program entry points only).

It is obvious that for some non-terminating programs the approximation sequence might be infinite, so that its limit cannot be automatically computed. The repartee to that problem consists in using induction, i.e. reasoning by recurrence on the length of the sequence (see 4.1.2).

Example

Suppose we want to prove that the optimal invariants of the program of fig. 6.4.a are given at fig 6.4.c.

By computing the first terms of the approximation sequence and next using some heuristics we must first discover the general term S(i) of the approximation sequence. We have found :

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

We must first show that this is a correct expression of the first terms of the ascending approximation sequence.

The initialization is :

$$P_i = \underline{\text{false}} \qquad i = 1..4$$

the first iteration is :

$$\begin{aligned}
P_1 &= (x = x_0) \ \underline{\text{and}} \ (y = y_0) \\
P_2 &= (P_1 \ \underline{\text{or}} \ P_3) \ \underline{\text{and}} \ (x \geq y) \\
&= (P_1 \ \underline{\text{or}} \ \underline{\text{false}}) \ \underline{\text{and}} \ (x \geq y) \\
&= P_1 \ \underline{\text{and}} \ (x \geq y) \\
&= (x = x_0) \ \underline{\text{and}} \ (y = y_0) \ \underline{\text{and}} \ (x_0 \geq y_0) \\
P_3 &= \{\exists x' \mid P_2(x') \ \underline{\text{and}} \ x = (x' - y)\} \\
&= (x_0 \geq y_0) \ \underline{\text{and}} \ (x' = x_0) \ \underline{\text{and}} \ (y = y_0) \ \underline{\text{and}} \ (x = x' - y) \\
&= (x_0 \geq y_0) \ \underline{\text{and}} \ (x = x_0 - y_0) \ \underline{\text{and}} \ (y = y_0) \\
P_4 &= (P_1 \ \underline{\text{or}} \ P_3) \ \underline{\text{and}} \ \underline{\text{not}} \ (x \geq y) \\
&= (x = x_0) \ \underline{\text{and}} \ (y = y_0) \ \underline{\text{and}} \ (x_0 < y_0)
\end{aligned}$$

These iterates are clearly of the general form S(i) for i = 0. For the induction step, supposing S(i) to be correct and replacing in the equations we must show that we obtain S(i + 1).

$$\begin{aligned}
P_1 &= (x = x_0) \ \underline{\text{and}} \ (y = y_0) \\
P_2 &= (P_1 \ \underline{\text{or}} \ P_3) \ \underline{\text{and}} \ (x \geq y) \\
&= (P_1 \ \underline{\text{and}} \ x \geq y) \ \underline{\text{and}} \ (P_3 \ \underline{\text{and}} \ (x \geq y)) \\
&= \{((x_0 \geq y_0) \ \underline{\text{and}} \ (x = x_0) \ \underline{\text{and}} \ (y = y_0)) \\
&\quad \underline{\text{or}} \\
&\quad \underset{j=0}{\overset{i}{\underline{\text{OR}}}} \ (\underset{k=0}{\overset{j}{\underline{\text{AND}}}} (x_0 - ky_0 \geq y_0) \ \underline{\text{and}} \ (x = x_0 - (j+1)y_0) \ \underline{\text{and}} \ (y = y_0))\} \\
&\quad \underline{\text{and}} \ \{x \geq y\} \\
&= ((x_0 \geq y_0) \ \underline{\text{and}} \ (x = x_0) \ \underline{\text{and}} \ (y = y_0)) \\
&\quad \underline{\text{or}} \\
&\quad \underset{j'=1}{\overset{i+1}{\underline{\text{OR}}}} \ (\underset{k=0}{\overset{j'}{\underline{\text{AND}}}} (x_0 - ky_0 \geq y_0) \ \underline{\text{and}} \ (x = x_0 - j'y_0) \ \underline{\text{and}} \ (y = y_0)) \\
&= \underset{j=0}{\overset{i+1}{\underline{\text{OR}}}} \ (\underset{k=0}{\overset{j}{\underline{\text{AND}}}} (x_0 - ky_0 \geq y_0) \ \underline{\text{and}} \ (x = x_0 - jy_0) \ \underline{\text{and}} \ (y = y_0))
\end{aligned}$$

$$P_3 = \{\exists x' \mid P_2(x') \text{ and } x = x'-y\}$$

$$= \underset{j=0}{\overset{i+1}{\text{OR}}} \{\exists x' \mid \underset{k=0}{\overset{j}{\text{AND}}} (x_0 - ky_0 \geq y_0) \text{ and } (x' = x_0 - jy_0) \text{ and } (y = y_0) \text{ and } (x=x'-y)\}$$

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

$$P_4 = (P_1 \text{ or } P_3) \text{ and } (x < y)$$

$$= (P_1 \text{ and } (x < y)) \text{ or } (P_3 \text{ and } (x < y))$$

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

or

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

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

Since now we have proved the general form S(i) of the i^{th} term of approximation sequence to be correct, the optimal invariants are obtained by :

$$S_{opt} = \lim_{i \to \infty} S(i)$$

which directly results in formulas of fig 6.4.c.

Discussion

It could be claimed that the users will have even more difficulties to find optimal invariants than they have to find approximate invariants. This may be true. Yet, discovery of invariants is necessary in both cases and it might turn out that the discovery of optimal invariants presents no more difficulties that the discovery of simply approximate invariants. The key to that problem is certainly symbolic execution.

6.8 Symbolic Execution

Symbolic execution is a widely used program analysis technique. (a.o., Burstall[1974], Cheatham and Townley[1976], Effigy[1975], Hantler and King [1976], Hoare[1976], King[1976], Select[1975], Sintzoff[1975], Yonezawa [1976]).

We will show that symbolic execution consists in solving a system of equations by successive approximations.

6.8.1 Abstract Contexts

With each program point will be associated an abstract context which is the set of different program paths :

$$\{p_1, \ldots, p_m\}$$

which may lead to that program point. Each program path p of $\{p_1, \ldots, p_m\}$ is of the form :

$$p = <(\pi = Q), (x_1 = E_1), \ldots, (x_i = E_i), \ldots, (x_n = E_n)>$$

The path condition π is equal to an assertion Q stating the conditions which had to be satisfied in order for that path to be executed.
The x_i are the program variables whereas the E_i are formal expressions depending on formal symbols $V_1, \ldots, V_n$ which represent the arbitrary initial values of the variables $x_1, \ldots, x_n$ on program entry.

In general, abstract contexts may have different representations, and rules of equivalence must be defined. From now on we will consider that abstract contexts are equal when formally equivalent.

In the following $p(\pi)$ denotes Q, $p(x_i)$ denotes E_i and $p(\alpha \leftarrow \beta)$ is a copy of p modified so that the value of α becomes β.

6.8.2 System of Equations

The dependance between abstract contexts associated to adjacent program points is defined by the following forward rules :

Entry points :

$$C_e = \{<(\pi = \underline{\text{true}}),\ (x_j = V_{e_j}),\ j = 1,\ \ldots,\ n>\}$$

where V_{e_j}, $j = 1, \ldots, n$ are different formal symbols wich represent the symbolic input values of the variables.

Assignment statements :

$$\{C\}$$
$$x := E$$
$$\{p(x \leftarrow E(x_i \leftarrow (p(x_i)),\ i = 1,\ \ldots,\ n)),\ \forall p \in C\}.$$

The variables x_i in the right-hand-side expression E are replaced by their values $p(x_i)$, (parenthesized to maintain the proper scope of operator) and the result is assigned to the left-hand-side variable x. (Algebraic simplifications are usually performed but in theory superfluous).

If statements :

$$\{C\}$$
$$\underline{\text{if}}\ B\ \underline{\text{then}}$$
$$\quad \{p(\pi \leftarrow (p(\pi)\ \underline{\text{and}}\ B(x_i \leftarrow (p(x_i))), i = 1,\ldots,n),\ \forall p \in C\}$$
$$\quad <\text{statement list}>$$
$$\underline{\text{fi}}$$

The path condition is updated for each possible path.

Labels and go to statements :

$$L : \{ \bigsqcup_{i \in \underline{\text{pred}}(L)} C_i \}$$

where $\underline{\text{pred}}(L)$ denotes the set of program points which may preceed L during any program execution.

The operation $\sqcup$ describes the union of two abstract contexts $C_1 = \{p_1, \ldots, p_m\}$ and $C_2 = \{q_1, \ldots, q_n\}$. $C_1 \sqcup C_2$ is the set $\{p_1, \ldots, p_m, q_1, \ldots, q_n\}$ where possible equivalent program paths are eliminated and replaced by the representant of the equivalence class.

6.8.3 Symbolic Execution Tree

We illustrate the application of the above rules to the program :

```
        {0}
              x := 1;
        {1}
loop :  {2}
              if x ≤ a then
        {3}
                    x := x+b ;
        {4}
                    goto loop ;
              fi;
        {5}
```

fig 6.8.3.a

The corresponding system of equations is :

$$\begin{cases} C_0 = \{<(\pi = \underline{true}),\ (x = \Omega),\ (a = \alpha),\ (b = \beta)>\} \\ C_1 = \{p(x \leftarrow 1),\ \forall p \in C_0\} \\ C_2 = C_1 \sqcup C_4 \\ C_3 = \{p(\pi \leftarrow p(\pi)\ \underline{and}\ (p(x) \leq p(a))),\ \forall p \in C_2\} \\ C_4 = \{p(x \leftarrow (p(x) + p(b)),\ \forall p \in C_3\} \\ C_5 = \{p(\pi \leftarrow p(\pi)\ \underline{and}\ (p(x) > p(a))),\ \forall p \in C_2\} \end{cases}$$

fig 6.8.3.b

Symbolic execution consists in computing the increasing sequence of successive approximations (2.4.1) :

Initialization :

$C_i = \{<\underline{false}, \perp, \perp, \perp>\}$

(we use the notation $<Q, \alpha, \beta, \gamma>$ to denote $<(\pi = Q), (x = \alpha), (a = \beta), (b = \gamma)>$)

Iteration 1 :

$C_0 = \{<\underline{true}, \Omega, \alpha, \beta>\}$

$C_1 = \{p(x \leftarrow 1),\ \forall p \in C_0\}$

$\quad = \{<\underline{true}, 1, \alpha, \beta>\}$

$C_2 = C_1 \sqcup C_4 = \{<\underline{true}, 1, \alpha, \beta>\} \cup \{<\underline{false}, \perp, \perp, \perp>\}$

$\quad = \{<\underline{true}, 1, \alpha, \beta>\}$

since the path condition (π = $\underline{false}$) describes an inaccessible path which needs not be considered.

$C_3 = \{p(\pi \leftarrow p(\pi)$ and $(p(x) \leq p(a))),\ \forall p \in C_2\}$
$= \{<(1 \leq \alpha),\ 1,\ \alpha,\ \beta>\}$
$C_4 = \{p(x \leftarrow (p(x)) + (p(b))),\ \forall p \in C_3\}$
$= \{<(1 \leq \alpha),\ 1+\beta,\ \alpha,\ \beta>\}$
$C_5 = \{p(\pi \leftarrow p(\pi)$ and $(p(x) > p(a))),\ \forall p \in C_2\}$
$= \{<(1 > \alpha),\ 1,\ \alpha,\ \beta>\}$

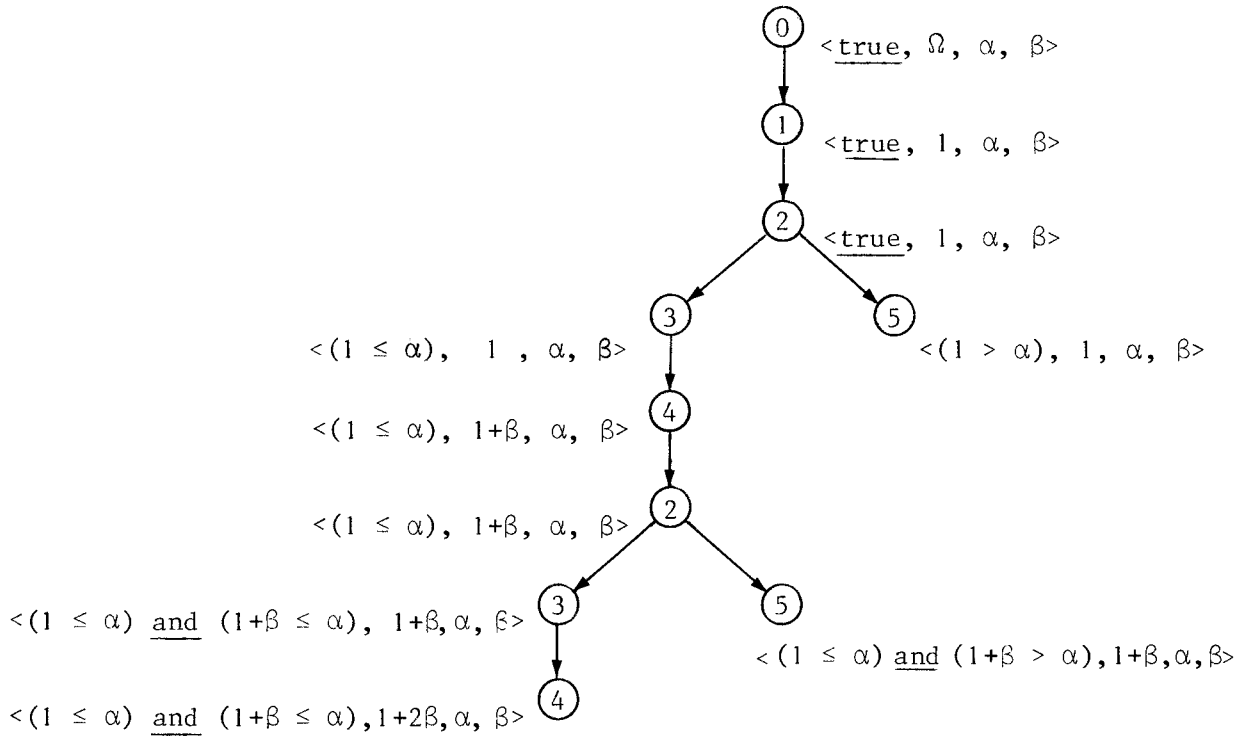
fig 6.8.3.c

Iteration 2 :

$C_2 = C_1 \sqcup C_4$
$= \{<$true$,\ 1,\ \alpha,\ \beta>,\ <(1 \leq \alpha),\ 1+\beta,\ \alpha,\ \beta>\}$
$C_3 = \{<(1 \leq \alpha),\ 1,\ \alpha,\ \beta>,\ <(1 \leq \alpha)$ and $(1+\beta \leq \alpha),\ 1+\beta,\ \alpha,\ \beta>\}$
$C_4 = \{<(1 \leq \alpha),\ 1+\beta,\ \alpha,\ \beta>,\ <(1 \leq \alpha)$ and $(1+\beta \leq \alpha),\ 1+2\beta,\ \alpha,\ \beta>\}$
$C_5 = \{<(1 > \alpha),\ 1, \alpha,\ \beta>,\ <(1 \leq \alpha)$ and $(1+\beta > \alpha),\ 1+\beta,\ \alpha,\ \beta>\}$

fig 6.8.3.d

So that at iteration 2 we have built the following symbolic execution tree, (Hantler and King[1976]) :



It is clear that we represented the abstract context C_i associated to program point i by the set of paths associated to each of the nodes labelled i in the above execution tree. Equivalently we could have represented the asbtract contexts associated with program point i by the maximal subtree (of the above symbolic execution tree) which leaves would have been labelled by i. Then the union of abstract 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 α and β this process would be infinite.

Thus symbolic execution must cope with the problem of passage to the limit in infinite sequences of successive approximations. In paragraph 4 we investigated the various mathematical techniques which permit to cope with infinite approximation sequences. Not surprisingly each of these mathematical techniques gave rise to program analysis techniques.

6.8.4 Verification of Properties of Optimal Symbolic Contexts

Scott's induction (4.3.3) consists in proving P(S) where $S = \mu(F)$ by the rule :

$$\{P(S_0) \text{ and } \{\{(\forall X)(P(X))\} \Longrightarrow \{P(F(X))\}\}\} \Longrightarrow \{P(S)\}$$

where S_0 is one of the first terms of the increasing sequence of successive approximations.

This approach is implicitly used in the technique of "cut-trees" of Hantler and King[1976].

Example

Let us prove the trivial fact that assertion $(x - b \leq a)$ holds at program point {4}, (see fig 6.8.3.a).

Base

After iteration 1 the abstract context C_4 is (see fig 6.8.3.c) :

$$C_4 = \{<(1 \leq \alpha),\ 1+\beta,\ \alpha,\ \beta>\}$$

Therefore trivially, $(C_4(x) - C_4(b) \leq C_4(a))$.

Induction Step

Suppose it is true of all paths in C_4 at iteration ℓ :

$$C_4(\ell) = \{<p_i,\ \gamma_i,\ \alpha_i,\ \beta_i>,\ i \in D\} \text{ with } \gamma_i - \beta_i \leq \alpha_i$$

replacing in the right members of the equations we get :

$$C_2(\ell+1) = C_4(\ell) \sqcup C_1(\ell)$$
$$= \{<\underline{\text{true}},\ 1,\ \alpha,\ \beta>,\ <p_i,\ \gamma_i,\ \alpha_i, \beta_i>,\ i \in D\}$$
$$C_3(\ell+1) = \{<(1 \leq \alpha),\ 1,\ \alpha,\ \beta>,\ <p_i \ \underline{\text{and}}\ (\gamma_i \leq \alpha_i),\ \gamma_i,\ \alpha_i,\ \beta_i>,\ i \in D\}$$
$$C_4(\ell+1) = \{<(1 \leq \alpha),\ 1+\beta,\ \alpha,\ \beta>,\ <p_i \ \underline{\text{and}}\ (\gamma_i \leq \alpha_i),\ \gamma_i+\beta_i,\ \alpha_i,\ \beta_i>,\ i \in D\}$$

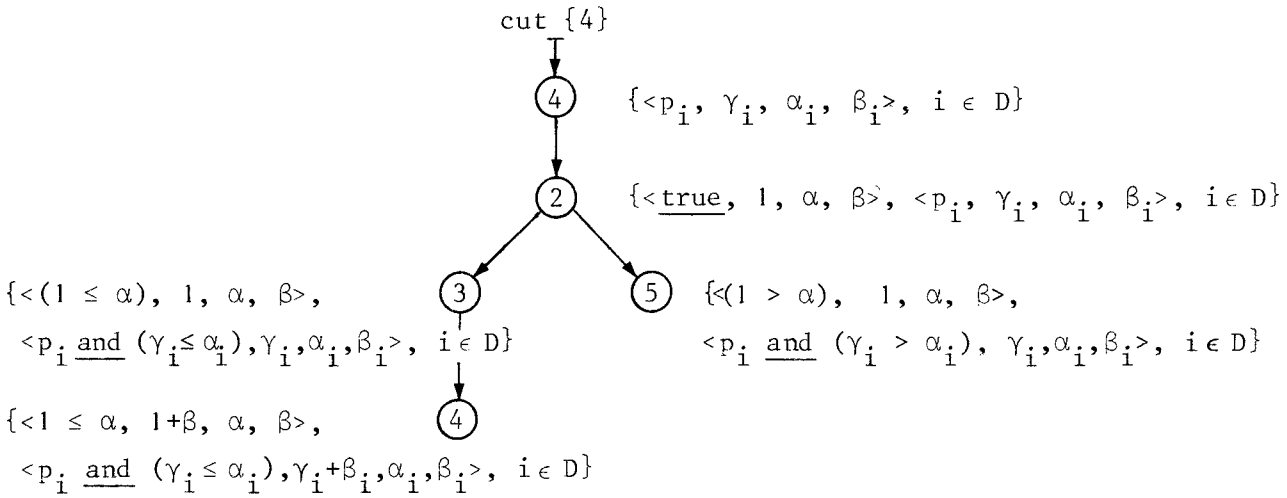
And now we have to prove $(x-b \leq a)$ for all paths of $C_4(\ell+1)$ that is both :

. $(1 \leq \alpha) \Longrightarrow (1+\beta-\beta \leq \alpha)$

. $\underbrace{(\gamma_i-\beta_i \leq \alpha_i)}_{\text{induction hypothesis}}$ and $\underbrace{(\gamma_i \leq \alpha_i)}_{\text{path condition}} \Longrightarrow (\gamma_i+\beta_i-\beta_i \leq \alpha_i)$

Notes

Notice that the induction step consisted in reasoning on the cut tree for {4}.



This approach is also that of Sintzoff[1975] although the joining paths operation ⊔ does not juxtapose the different paths but rather merges them using "alternative expressions" instead of simple formal values. This simplifies the description of the tree of considered alternatives since common parts of the different alternatives are merged together. Moreover the technique is applied to the verification of procedures.

6.8.5 Verification of the Correctness of Optimal Symbolic Contexts

According to 4.1.2 we can discover the limit S_∞ of the increasing approximation sequence $S_0, \ldots, S_\ell, \ldots$ by using (asking to the programmer) the general form $f(\ell)$ of S_ℓ. This consists in proving the correctness of $S_\ell = f(\ell)$ by showing that :

- $S_0 = f(0)$ (or equivalently $S_1 = f(1)$)
- $f(\ell+1) = F(f(\ell))$

and next passing to the limit by

$$S_\infty = \lim_{\ell \to \infty} f(\ell)$$

This mathematical technique is essentially the one used by Burstall [1974].

Example

In our example of fig 6.8.3.a, $f(\ell)$ is entirely defined by :

$$C_2(\ell) = \{<\overline{\underset{i=0}{\overset{j-1}{\text{AND}}}}\ (1 + i\beta \leq \alpha),\ 1+j\beta,\ \alpha,\ \beta>,\ j = 0, \ldots,\ \ell-1\}$$

with the convention that $\overline{\underset{i\in\Delta}{\text{AND}}}\ (e_i)$ is true for an empty indexing set $\Delta = \emptyset$.

Replacing in the system of equations of fig 6.8.3.b we get :

$$C_3(\ell) = \{<\overline{\underset{i=0}{\overset{j}{\text{AND}}}}\ (1+i\beta \leq \alpha), 1+j\beta,\ \alpha,\ \beta>,\ j = 0,\ \ldots, \ell\ -1\}$$

$$C_4(\ell) = \{<\overline{\underset{i=0}{\overset{j}{\text{AND}}}}\ (1+i\beta \leq \alpha), 1+(j+1)\beta,\ \alpha,\ \beta>,\ j = 0, \ldots,\ \ell-1\}$$

$$C_2(\ell+1) = C_4(\ell) \sqcup C_1(\ell) = C_4(\ell) \cup \{<\underline{\text{true}},\ 1,\ \alpha,\ \beta>\}$$

$$= \{<\underline{\text{true}},\ 1,\ \alpha,\ \beta>,$$

$$<\underline{\overset{j'-1}{\underset{i=0}{\text{AND}}}}\ (1+i\beta \le \alpha),\ 1+j'\beta,\ \alpha,\ \beta>,\ j' = 1,\ldots,\ell\}$$

$$= \{<\underline{\overset{j-1}{\underset{i=0}{\text{AND}}}}\ (1+i\beta \le \alpha),\ 1+j\beta,\ \alpha,\ \beta>,\ j = 0,\ldots,\ell\}$$

which is of the form used as hypothesis of the induction step and therefore is correct. It is obvious that in iteration 2 we discover $C_2(2)$ (see fig 6.8.3.d).

Now the final result of the symbolic execution is obtained as

$$\lim_{\ell\to\infty} f(\ell)$$

and we get the result :

$$C_0 = \{<\underline{\text{true}},\ \Omega,\ \alpha,\ \beta>\}$$

$$C_1 = \{<\underline{\text{true}},\ 1,\ \alpha,\ \beta>\}$$

$$C_2 = \{<\underline{\overset{j-1}{\underset{i=0}{\text{AND}}}}\ (1+i\beta \le \alpha),\ 1+j\beta,\ \alpha,\ \beta>,\ \forall j \ge 0\}$$

$$C_3 = \{<\underline{\overset{j}{\underset{i=0}{\text{AND}}}}\ (1+i\beta \le \alpha),\ 1+j\beta,\ \alpha,\ \beta>,\ \forall j \ge 0\}$$

$$C_4 = \{<\underline{\overset{j}{\underset{i=0}{\text{AND}}}}\ (1+i\beta \le \alpha),\ 1+(j+1)\beta,\ \alpha,\ \beta>,\ \forall j \ge 0\}$$

$$C_5 = \{<(\underline{\overset{j-1}{\underset{i=0}{\text{AND}}}}\ (1+i\beta \le \alpha))\ \underline{\text{and}}\ (1+j\beta > \alpha),\ 1+j\beta,\ \alpha,\ \beta>,\ \forall j \ge 0\}$$

fig 6.8.5.a

Note

Since we have the optimal solution to the equations we can prove or disprove termination (see 6.4). For example the program of fig 6.8.3.a does not terminate over the input specification $\Phi(\alpha, \beta) = (\beta = 0,\ \alpha \ge 1)$ since

$$(\forall(\alpha, \beta) \mid \Phi(\alpha, \beta)),\ \exists h \mid (\forall p \in C_h,\ C_h(\pi) = \underline{\text{false}}).$$

Taking h = {5}, and C_5 as defined at fig 6.8.5.a we have :

$$C_5 = \{<(\underline{\overset{j-1}{\underset{i=0}{AND}}}\ (1+i\beta \leq \alpha))\ \underline{and}\ (1+j\beta > \alpha),\ ...>,\ \forall j \geq 0\}$$

under the condition $\beta = 0$ and $\alpha \geq 1$ the path conditions of all paths of C_5 are false, and therefore these paths are inaccessible. Hence when $\{\beta = 0, \alpha \geq 1\}$, the program cycles.

Note also that Burstall[1974] does not use induction on the length of the approximation sequence but instead structural induction. Moreover he does not require $f(\ell)$ to correspond the least fixpoint of the symbolic equations so that he cannot prove termination.

6.8.6 Discovery of the Optimal Symbolic Contexts

The last mathematical technique we will investigate to cope with infinite symbolic execution trees is 4.2.3 since it permits the *discovery* of the optimal symbolic contexts as opposed to simple *verifications* in the previous paragraph. Recall that the recurrence relationship $S_0 = \perp^k$ and $S_{\ell+1} = F(S_\ell)$ between consecutive terms of the increasing approximation sequence leads to a system of difference equations, which may be solved to get S_ℓ as a function $f(\ell)$ of ℓ, which gives the least solution S_∞ as $\lim_{\ell \to \infty} (f(\ell))$.

This approach is implicitely used in Grief and Waldinger[1974], in Cheatham and Townley[1976] and in the "algorithmic approach" of Katz and Manna[1976], etc.

Example

Let us apply this technique to the program given at fig 6.8.3.a.

Since all contexts in the system of equations fig 6.8.3.b depend on C_2, we will try to discover the value of C_2 in the first place.

A first iteration of the ascending approximation sequence corresponding to the equations fig 6.8.3.b leads to the basis for difference equations.

$C_2(1) = \{<\underline{true}, 1, \alpha, \beta>\} = C_1$ which is constant. (See fig 6.8.3.c and notice that a path with a path condition equal to false is inaccessible and therefore needs not be considered).

Then we establish a recurrence relationship between $C_2(\ell+1)$ and $C_2(\ell)$ using the equations fig 6.8.3.b :

$$C_2(\ell) = \{<p_{i,\ell}, \gamma_{i,\ell}, \alpha_{i,\ell}, \beta_{i,\ell}>, i \in D(\ell)\}$$

$$C_3(\ell) = \{<p_{i,\ell} \underline{\text{and}} (\gamma_{i,\ell} \leq \alpha_{i,\ell}), \gamma_{i,\ell}, \alpha_{i,\ell}, \beta_{i,\ell}>, i \in D(\ell)\}$$

$$C_4(\ell) = \{<p_{i,\ell} \underline{\text{and}} (\gamma_{i,\ell} \leq \alpha_{i,\ell}), \gamma_{i,\ell} + \beta_{i,\ell}, \alpha_{i,\ell}, \beta_{i,\ell}>, i \in D(\ell)\}$$

$$C_2(\ell+1) = C_1(\ell) \sqcup C_4(\ell)$$

$$= C_1 \sqcup C_4(\ell) \quad \text{since } C_1 \text{ is constant.}$$

Notice that $C_4(\ell) = \overline{\Delta}(C_2(\ell))$

where $\overline{\Delta}\{<p_{i,\ell}, \gamma_{i,\ell}, \alpha_{i,\ell}, \beta_{i,\ell}>, i \in D\}$

$$= \{\Delta(<p_{i,\ell}, \gamma_{i,\ell}, \alpha_{i,\ell}, \beta_{i,\ell}>), i \in D\}$$

and

$$\Delta(<p_{i,\ell}, \gamma_{i,\ell}, \alpha_{i,\ell}, \beta_{i,\ell}>)$$

$$= <p_{i,\ell} \underline{\text{and}} (\gamma_{i,\ell} \leq \alpha_{i,\ell}), \gamma_{i,\ell} + \beta_{i,\ell}, \alpha_{i,\ell}, \beta_{i,\ell}>$$

thus

$$C_2(\ell+1) = C_1 \sqcup \overline{\Delta}(C_2(\ell))$$

Hence the recurrence relations defining C_2 are :

$$\begin{cases} C_2(1) = C_1 \\ C_2(\ell+1) = C_1 \sqcup \overline{\Delta}(C_2(\ell)) \end{cases}$$

which can be solved directly, using the property that $\overline{\Delta}$ is distributive over $\sqcup$, this yields :

$$C_2(\ell) = \bigsqcup_{i=0}^{\ell-1} \overline{\Delta}^i(C_1)$$

$$= \{\Delta^i(<\underline{true}, 1, \alpha, \beta>), i = 0, \ldots, \ell-1\}$$

It remains now to determine the multivalued function Δ^i. This is done using the fact that Δ^i is defined by recurrence : $\Delta^{i+1} = \Delta \circ \Delta^i$ and Δ^0 is the identity function.

We have :

$$\Delta^0(\langle \underline{\text{true}}, 1, \alpha, \beta \rangle)$$
$$= \langle \underline{\text{true}}, 1, \alpha, \beta \rangle$$
$$= \langle p_0, \gamma_0, \alpha_0, \beta_0 \rangle$$

Let $\Delta^i(\langle \underline{\text{true}}, 1, \alpha, \beta \rangle)$
be $\langle p_i, \gamma_i, \alpha_i, \beta_i \rangle$

then $\Delta^{i+1}(\langle \underline{\text{true}}, 1, \alpha, \beta \rangle)$
$$= \Delta(\langle p_i, \gamma_i, \alpha_i, \beta_i \rangle)$$
$$= \langle p_i \ \underline{\text{and}} \ (\gamma_i \leq \alpha_i), \gamma_i + \beta_i, \alpha_i, \beta_i \rangle$$
$$= \langle p_{i+1}, \gamma_{i+1}, \alpha_i, \beta_i \rangle$$

These recurrence relations may be solved directly, yielding :

$$\alpha_0 = \alpha \ \underline{\text{and}} \ \alpha_{i+1} = \alpha_i \Longrightarrow \alpha_i = \alpha$$
$$\beta_0 = \beta \ \underline{\text{and}} \ \beta_{i+1} = \beta_i \Longrightarrow \beta_i = \beta$$
$$\gamma_0 = 1 \ \underline{\text{and}} \ \gamma_{i+1} = \gamma_i + \beta_i$$
$$\Longrightarrow \gamma_{i+1} = \gamma_i + \beta$$
$$\Longrightarrow \gamma_i = 1 + i\beta$$

$$(p_0 = \underline{\text{true}}) \ \underline{\text{and}} \ (p_{i+1} = p_i \ \underline{\text{and}} \ (\gamma_i \leq \alpha_i))$$
$$\Longrightarrow p_{i+1} = p_i \ \underline{\text{and}} \ (1 + i\beta \leq \alpha)$$
$$\Longrightarrow p_i = \overset{i-1}{\underset{j=0}{\underline{\text{AND}}}} \ (1 + j\beta \leq \alpha)$$

we have found :

$$\Delta^i(\langle \underline{\text{true}}, 1, \alpha, \beta \rangle)$$
$$= (\langle \overset{i-1}{\underset{j=0}{\underline{\text{AND}}}} \ (1 + j\beta \leq \alpha), 1+i\beta, \alpha, \beta \rangle)$$

and therefore

$$C_2(\ell) = \{\Delta^i(<\underline{\text{true}},\ 1,\ \alpha,\ \beta>),\ i = 0,\ldots,\ \ell-1\}$$

$$= \{<\underline{\text{AND}}_{j=0}^{i-1}\ (1+j\beta \leq \alpha),\ 1+i\beta,\ \alpha,\ \beta>,\ i = 0,\ldots,\ \ell-1\}$$

Thus the optimal solution to equations fig 6.8.3.b is found to be :

$$\lim_{\ell\to\infty} (C_2(\ell))$$

$$= \{<\underline{\text{AND}}_{j=0}^{i-1}\ (1+j\beta \leq \alpha),\ 1+i\beta,\ \alpha,\ \beta>,\ \forall i \geq 0\}$$

The other symbolic contexts are straightforwardly obtained replacing $C_2(\infty)$ by its value in the equations. (One can verify that this corresponds to the solution proved to be correct in fig 6.8.5.a but this is useless here since the discovered result is guaranteed to be correct).

6.8.7 A Note On Loop Counters

Notice that in the previous paragraphs the basis for the recurrence was the length of the ascending approximation sequence, whereas in the methods cited in example one uses (eventually dummy) loop counters.

These dummy counters are used to denote relations among the number of times various paths have been executed and to help express the values assumed by the program variables. This is acceptable for simple programs with simply nested loops for which there exists a trivial relationship between the counters and the rank in the approximation sequence. Yet when considering program with inextricably intermixed loops the rank in the approximation sequence is no longer a simple function of the loop counters, hence the use of counters fails (see a.o. Greif and Waldinger[1974] page 114, Katz and Manna[1976], p. 205).

The reasoning on the approximation sequence should be the parade to these problems, and in fact should be used in all program analysis techniques.

6.9 *Concluding Remarks*

Program partial correctness proving methods are non-constructive since they only permit to verify that an approximate solution S of the system of equations P = F(P) is correct. The criterion of validity for S is that S_{opt} ==> S where S_{opt} is the optimal set of invariants defined as the least fixpoint of F. This criterion has been shown to be equivalent to S <== F(S).

Heuristic methods may be used to improve S. (Wegbreit[1974], Katz and Manna[1976]). They are based on the fact that if S_{opt} => S then S_{opt}=> F(S), and F(S) is generally better than S, (backward equations are also widely used (see 7)). More generally heuristic methods consist in finding an approximate solution of the system of equations by strengthening the iterates of the approximation sequence (see 4.4).

Symbolic execution is a constructive method to discover program properties since in fact it consists in computing the approximation sequence which converges to the optimal solution of the equations. However the general problem of finding an algorithm to generate optimal invariants for any program is unsolvable. Otherwise stated approximation sequences may be infinite. However some methods (see 4) permit to directly pass to the limit and obtain the optimal invariants. Thus symbolic execution appears to be a promising technique. Moreover it is the natural method usually chosen by programmers to hand-prove their programs.

7. APPLICATION TO THE DENOTATIONAL SEMANTICS OF PROGRAMMING LANGUAGES

The mathematical, denotational, fixpoint or topological semantics was introduced by Scott and Strachey (Scott[1970], Scott and Strachey[1971]) and further developped by several authors. A very helpful guide to the literature may be found is Scott[1976]. (This fundamental paper provides a close

and rigorous look at mathematical foundations which details have been voluntarily omitted here).

7.1 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 :

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

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 $\overline{X}_0$, their final values will be $F_P(\overline{X}_0)$ 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 $F(\overline{X}) = \bot$ whenever $F(\overline{X})$ is undefined for $\overline{X} \in D^n$. Moreover we naturally extend F to $D^{n'} \to D^{n'}$ by defining $F(\bot) = \bot$ and $F(\top) = \top$.

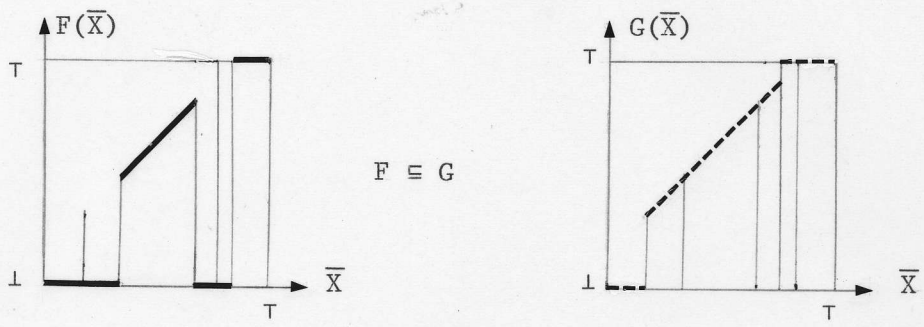
Let us now define an ordering $\sqsubseteq$ among functions in $D^{n'} \to D^{n'}$ by :

$$\{F \sqsubseteq G\} \Longleftrightarrow \{\forall \overline{X} \in D^{n'}, \ F(\overline{X}) \sqsubseteq_{D^{n'}} G(\overline{X})\}$$

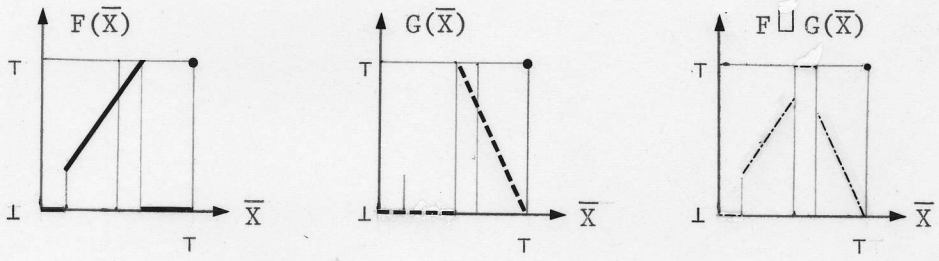
and let $\sqcup$ be the corresponding least upper bound operation.

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 ordering $\sqsubseteq_{D^{n'}}$ on $D^{n'} = [0, 1]$ is not the usual one ($\leq$) since any two distinct points of D^n are not comparable.

An example of comparable functions F and G would be



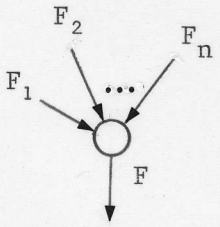
An example of union of (non-comparable) functions F and G would be :



7.2 *Functional Equations*

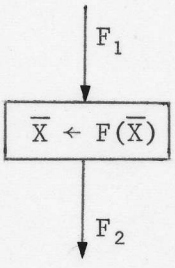
Let us now define the syntactic mechanism which permits to associate a system of equations with any flowchart program.

Junction nodes :



$$F = \coprod_{i=1}^{n} F_i$$

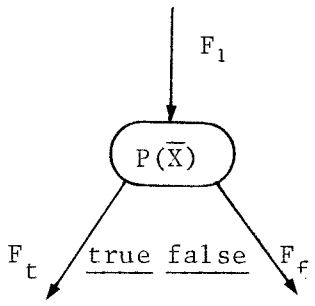
Assignment nodes :



$$F_2 = F \circ F_1$$

We consider a parallel assignment of n values $F(\overline{X})$ to the n variables $\overline{X}$. The semantics is given by functional composition denoted $\circ$.

Test nodes



$$F_t = (\iota \mid P) \circ F$$

$$F_f = (\iota \mid \underline{not}\ P) \circ F$$

We note 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 ($P : D^{n'} \to \{\underline{true}, \underline{false}\}$), we note $(F \mid P)$ the restriction of the function F to the subset of $D^{n'}$ satisfying the predicate P, therefore :

$$\forall \overline{X} \in D^{n'}$$

$$(F \mid P)(\overline{X}) = \underline{if}\ P(\overline{X})\ \underline{then}\ F(\overline{X})\ \underline{else}\ \perp\ \underline{fi}$$

The semantics of a test is simply the restriction of the input function to the domain satisfying the test.

Let us now quote some useful properties of the functional operations $\circ$ (composition), $\sqcup$ (union) and $\mid$ (restriction).

We define $\perp_{D^{n'} \to D^{n'}}$ to be the constant function which result is always the infimum $\perp$ of $D^{n'}$.
As before we note $F^0 = \iota$ and $F^{n+1} = F \circ F^n$.

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

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

$$(F \mid \underline{true}) = F$$

$$(F \mid \underline{false}) = \perp_{D^{n'} \to D^{n'}}$$

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

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

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

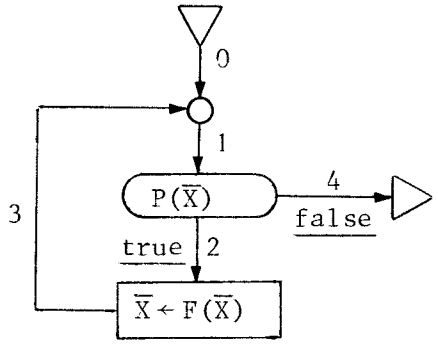
$$F \circ \left(\bigsqcup_{i \in \Delta} G_i \right) = \bigsqcup_{i \in \Delta} (F \circ G_i)$$

7.3 Example of the Semantics of While Loops

A while loop such as :

$$\text{while } P(\overline{X}) \text{ do } \overline{X} \leftarrow F(\overline{X}) \text{ od}$$

over the set of variables $\overline{X}$ is a syntactic denotation of the flowchart :



Its semantics is given by the least fixpoint of the system of equations :

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

The least fixpoint is the limit of the ascending approximation sequence also called Kleene's sequence in that particular application (Kleene[1952], first recursion theorem) :

Initialization :

$$F_i(0) = \bot_{D^{n'} \to D^{n'}} \qquad i = 0 \,..\, 4$$

Step 1 :

$$F_0(1) = \iota$$

$$F_1(1) = F_0(1) \sqcup F_3(0) = \iota \sqcup \bot_{D^{n'}, D^{n'}} = \iota$$

$$F_2(1) = (\iota \mid P) \circ F_1(1) = (\iota \mid P) \circ \iota = (\iota \mid P)$$

$$F_3(1) = F \circ F_2(1) = F \circ (\iota \mid P) = (F \mid P)$$

$$F_4(1) = (\iota \mid \underline{\text{not}}\ P) \circ F_1(1) = (\iota \mid \underline{\text{not}}\ P) \circ \iota = (\iota \mid \underline{\text{not}}\ P)$$

Step 2 :

$$F_0(2) = \iota$$

$$F_1(2) = F_0(2) \sqcup F_3(1)$$
$$= \iota \sqcup (F \mid P)$$

$$F_2(2) = (\iota \mid P) \circ F_1(2)$$
$$= (\iota \mid P) \circ (\iota \sqcup (F \mid P))$$
$$= ((\iota \mid P) \circ \iota) \sqcup ((\iota \mid P) \circ (F \mid P))$$
$$= (\iota \mid P) \sqcup (F \mid P \ \underline{\text{and}}\ P \circ F)$$

$$F_3(2) = F \circ F_2(2)$$
$$= F \circ ((\iota \mid P) \sqcup (F \mid P \ \underline{\text{and}}\ P \circ F))$$
$$= (F \mid P) \sqcup (F^2 \mid P \ \underline{\text{and}}\ P \circ F)$$

$$F_4(2) = (\iota \mid \underline{\text{not}}\ P) \circ F_1(2)$$
$$= (\iota \mid \underline{\text{not}}\ P) \circ (\iota \sqcup (F \mid P))$$
$$= ((\iota \mid \underline{\text{not}}\ P) \circ \iota) \sqcup ((\iota \mid \underline{\text{not}}\ P) \circ (F \mid P))$$
$$= (\iota \mid \underline{\text{not}}\ P) \sqcup (F \mid P \ \underline{\text{and}}\ \underline{\text{not}}\ P \circ F)$$

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

Step j :

$$F_0(j) = \iota$$

$$F_1(j) = \bigsqcup_{k=0}^{j-1} (F^k \mid \underline{\text{AND}}_{i=0}^{k-1} P \circ F^i)$$

$$F_2(j) = \bigsqcup_{k=0}^{j-1} (F^k \mid \underset{i=0}{\overset{k}{\underline{\text{AND}}}} P \circ F^i)$$

$$F_3(j) = \bigsqcup_{k=0}^{j-1} (F^{k+1} \mid \underset{i=0}{\overset{k}{\underline{\text{AND}}}} P \circ F^i)$$

$$F_4(j) = \bigsqcup_{k=0}^{j-1} (F^k \mid (\underset{i=0}{\overset{k-1}{\underline{\text{AND}}}} P \circ F^i) \underline{\text{and}} (\underline{\text{not}}\ P \circ F^k))$$

These may then be proved to be correct using mathematical induction (4.1.2) :

- It is first easy to verify that the above formulas are correct for j = 0,1 and 2 with the following usual conventions :

  $\bigsqcup_{i \in \Delta} F_i = \bot_{D^{n'} \to D^{n'}}$ when the indexing set Δ is empty

  $\underset{i \in \Delta}{\overset{}{\underline{\text{AND}}}} P_i = \underline{\text{true}}$ 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 :

$$F_0(j+1) = \iota$$

$$F_1(j+1) = F_0(j+1) \sqcup F_3(j)$$

$$= \iota \sqcup (\bigsqcup_{k=0}^{j-1} (F^{k+1} \mid \underset{i=0}{\overset{k}{\underline{\text{AND}}}} P \circ F^i))$$

$$= \iota \sqcup (\bigsqcup_{k'=1}^{j} (F^{k'} \mid \underset{i=0}{\overset{k'-1}{\underline{\text{AND}}}} P \circ F^i))$$

$$= \bigsqcup_{k=0}^{(j+1)-1} (F^k \mid \underset{i=0}{\overset{k-1}{\underline{\text{AND}}}} P \circ F^i)$$

$$F_2(j+1) = (\iota \mid P) \circ F_1(j+1)$$

$$= (\iota \mid P) \circ \bigsqcup_{k=0}^{j} (F^k \mid \underset{i=0}{\overset{k-1}{\underline{\text{AND}}}} P \circ F^i)$$

$$= \bigsqcup_{k=0}^{j} (\iota \mid P) \circ (F^k \mid \underset{i=0}{\overset{k-1}{\underline{\text{AND}}}} P \circ F^i))$$

$$= \bigsqcup_{k=0}^{j} (\iota \circ F^k \mid (P \circ F^k) \underline{\text{and}} (\underset{i=0}{\overset{k-1}{\underline{\text{AND}}}} P \circ F^i))$$

$$= \bigsqcup_{k=0}^{(j+1)-1} (F^k \mid \underset{i=0}{\overset{k}{\underline{\text{AND}}}} P \circ F^i)$$

$$F_3(j+1) = F \circ F_2(j+1)$$

$$= F \circ \left(\coprod_{k=0}^{j} (F^k \mid \underline{\underset{i=0}{\overset{k}{\text{AND}}}}\ P \circ F^i) \right)$$

$$= \coprod_{k=0}^{j} (F \circ (F^k \mid \underline{\underset{i=0}{\overset{k}{\text{AND}}}}\ P \circ F^i))$$

$$= \coprod_{k=0}^{(j+1)-1} (F^{k+1} \mid \underline{\underset{i=0}{\overset{k}{\text{AND}}}}\ P \circ F^i)$$

$$F_4(j+1) = (\iota \mid \underline{\text{not}}\ P) \circ F_1(j+1)$$

$$= (\iota \mid \underline{\text{not}}\ P) \circ \coprod_{k=0}^{j} (F^k \mid \underline{\underset{i=0}{\overset{k-1}{\text{AND}}}}\ P \circ F^i)$$

$$= \coprod_{k=0}^{j} (\iota \mid \underline{\text{not}}\ P) \circ (F^k \mid \underline{\underset{i=0}{\overset{k-1}{\text{AND}}}}\ P \circ F^i)$$

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

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

The general term of the (Gauss-Seidel transformed 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 :

$$\lim_{j \to \infty} F_4(j)$$

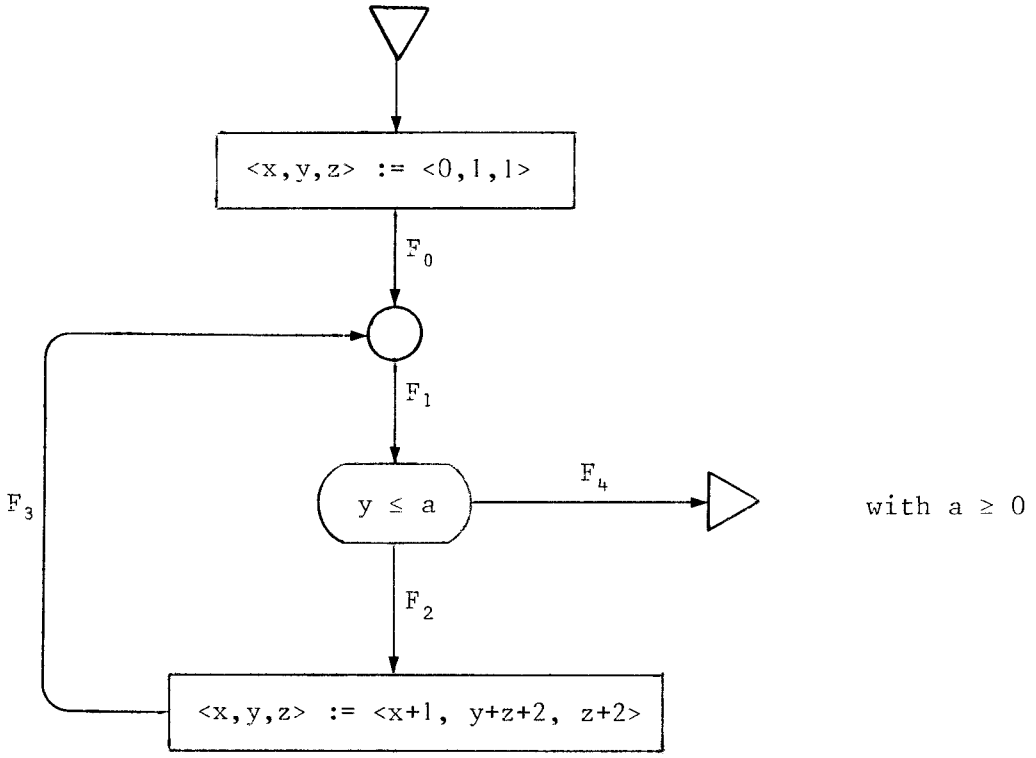
which is :

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

7.4 *Application to a Program for Computing* $\lfloor\sqrt{a}\rfloor$

Before proceeding it seems wise to verify that our formal definition of the semantics of the while schema indeed captures our intuitive understanding of the schema. Let us consider an example of interpretation of this schema which is taken from Manna and Vuillemin[1972] :

The program is the following :



it computes the integer square root $\lfloor\sqrt{a}\rfloor$ of a natural integer "a" using the arithmetic property

$$\forall n \in \mathbb{N}, \quad 1 + 3 + \ldots + 2n-1 = n^2$$

The semantics of this program is the function

$$R = \underline{\text{while}} \circ \mu$$

where

$$\mu(\langle x,y,z\rangle) = \langle 0,1,1\rangle$$

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

$$P(\langle x,y,z\rangle) = (y \leq a)$$

$$F(\langle x,y,z\rangle) = \langle x+1,\ y+z+2,\ z+2\rangle$$

We have :

$$R = \underline{\text{while}} \circ \mu$$

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

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

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

Let us first compute $F^n \circ \mu$ for $n \geq 0$.

$F^0 \circ \mu = 1 \circ \mu = \mu = \langle 0,1,1 \rangle$

induction hypothesis :

$F^j \circ \mu = \langle j, (j+1)^2, 2j+1 \rangle$

$F^{j+1} \circ \mu = \langle j+1,\ (j+1)^2 + 2j + 1 + 2,\ 2j + 1 + 2 \rangle$

$= \langle j+1,\ ((j+1) + 1)^2,\ 2(j+1) + 1 \rangle$

by recurrence :

$$F^n \circ \mu = \langle n,\ (n+1)^2,\ 2n+1 \rangle,\ \forall n \geq 0$$

and also

$$P \circ F^n \circ \mu = (n+1)^2 \leq a$$

Replacing in "R" we get :

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

Simplifying using the arithmetic property :

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

we obtain :

$$R = \coprod_{k=0}^{\infty} (\langle k,\ (k+1)^2,\ 2k+1 \rangle \mid k^2 \leq a < (k+1)^2)$$

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

$$R = \left(\coprod_{k \neq \lfloor \sqrt{a} \rfloor} (\langle k,\ (k+1)^2,\ 2k+1 \rangle \mid \underline{\text{false}}) \right) \sqcup (\langle \lfloor \sqrt{a} \rfloor, (\lfloor \sqrt{a} \rfloor + 1)^2, 2\lfloor \sqrt{a} \rfloor + 1 \rangle \mid \underline{\text{true}})$$

$$R = \left(\coprod_{k \neq \lfloor \sqrt{a} \rfloor} \perp_{D^{n'} \to D^{n'}} \right) \sqcup \left(\langle \lfloor \sqrt{a} \rfloor,\ (\lfloor \sqrt{a} \rfloor + 1)^2,\ 2\lfloor \sqrt{a} \rfloor + 1 \rangle \right)$$

$$R = \langle \lfloor \sqrt{a} \rfloor,\ (\lfloor \sqrt{a} \rfloor + 1)^2,\ 2\lfloor \sqrt{a} \rfloor + 1 \rangle$$

Which is the expected result of the program.

7.5 Concluding 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

a - "Static" or implicit definition :

The function computed by the while command is the term $F_4(\infty)$ of the least solution $\langle F_1(\infty), F_4(\infty) \rangle$ of the functional equations :

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

b - "Dynamic" or explicit definition :

The function computed by the while command is :

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

The question of which of the two (equivalent) definitions is the most useful for expressing the semantics of while commands is a polemical one (Dijkstra[1976], Hehner[1976]). However both approaches are "dual" 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 $F = m \frac{dv}{dt}$) and a static law expressing the conservation of some quantity (e.g. conservation of the quantity of movement mv).

Remark 2 : (forward and backward equations).

The semantics of the while loop as proposed by Manna and Vuillemin [1972] is the least fixpoint of the fonctional φ defined by the equation :

$$\varphi(X) = \underline{\text{if}}\ P(\overline{X})\ \underline{\text{then}}\ \varphi(F(\overline{X}))\ \underline{\text{else}}\ \underline{\text{null}}\ \underline{\text{fi}}$$

in our formalism :

$$\varphi = (\varphi \circ F \mid P) \sqcup (\iota \mid \underline{\text{not}}\ P)$$

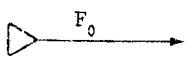
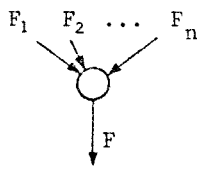
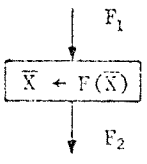
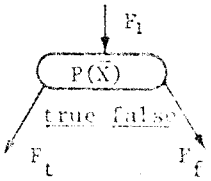
which solution

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

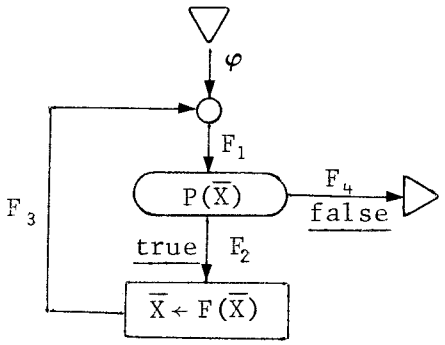
is the same as ours.

However this equation is obtained by the backward rules of Mc Carthy[1963] (see Manna[1974], pp. 324-326), whereas we used forward equations.

These two alternative methods may be compared in the following table :

Basic constructs	Forward rules	Backward rules
Entry node 	$F_0 = \iota$	
Junction node 	$F = \coprod_{i=1}^{n} F_i$	$F_i = F$ $i = 1 .. n$
Assignment node 	$F_2 = F \circ F_1$	$F_1 = F_2 \circ F$
Test nodes 	$F_t = (F_1 \mid P)$ $F_f = (F_1 \mid \underline{\text{not}}\ P)$	$F_1 = (F_t \mid P) \sqcup (F_f \mid \underline{\text{not}}\ P)$
Exit nodes 		$F_e = \iota$

Applying the backward rules to the while loop schema we get :



$$\varphi = F_1$$

$$F_3 = F_1$$

$$F_1 = (F_2 \mid P) \sqcup (F_4 \mid \underline{\text{not}}\ P)$$

$$F_2 = F_3 \circ F$$

$$F_4 = \iota$$

which simplifies in :

$$\varphi = (\varphi \circ F \mid P) \sqcup (\iota \mid \underline{\text{not}}\ P)$$

The uses of forward or backward equations are generally a matter of taste (Dijkstra[1976], page 214).

8. APPLICATION TO COMPILER VERIFICATION OR DISCOVERY OF PROGRAM PROPERTIES

The most familiar abstract interpretation performed by compilers is static type checking.

When the number of types involved in a program is finite the compiler can solve the system of type equations using a finite approximation sequence (4.2.1). However most languages permit to write programs using a potentially infinite number of types. The compiler must then cope with infinite approximation sequences. The two most common alternatives are :

- Type verification (4.1.1) : by means of declarations the programmer provides a solution or a correct approximation of the exact solution of the system of equations. The compiler simply verifies that the declarations of the programmer are correct (Ledgard[1972], Algol 68[1976]).

- Type discovery (4.4) : the programmer needs not declare the type of the objects manipulated by its program (APL, SETL), or needs only declare gross global type properties which are not always sufficient for local type checking (PASCAL, see Cousot[1977b]). The compiler must then discover a correct approximation of the exact solution of the system of type equations (Cousot[1976], Tenenbaum[1974]).

We now illustrate discovery of program properties using a finite and then an infinite interpretation space. The general idea is to partition the potential values space of each variable in a finite or infinite number of subsets, each being represented by an abstract value. At each program point the problem is to determine to which subsets will belong the dynamic values of each variable. This is done by solving a system of equations which is obtained by interpreting the program basic operations and tests as acting on the abstract values of their operands. This static analysis of program properties resembles type discovery yet the space of abstract values may introduce a partition of the values space which is thiner than the one induced by conventional types. Accordingly the building of the system of equations may involve a deeper analysis of the semantics of primitive operations.

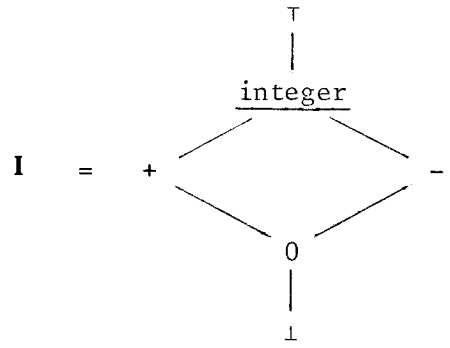
8.1 Finite Abstract Evaluation of Programs

Examples of finite abstract evaluation of programs may be found in type checking (Naur[1966]), checking of sufficient conditions for program properties to be satisfied (Sintzoff[1972]), finite state program testing (Henderson and Quarendon[1974], Henderson[1975]), elimination of unnecessary copying operations (Schwartz[1975]), etc.

8.1.1 Example of an Abstract Interpretation of Integers, Stacks and Trees

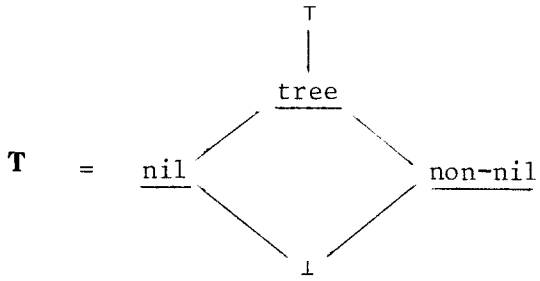
In order to give an extremely simple example of finite abstract interpretation, let us consider a programming language allowing the definition of data types using the basic integer type and the stack and binary tree constructors.

Let us partition the set of integer values as follows :



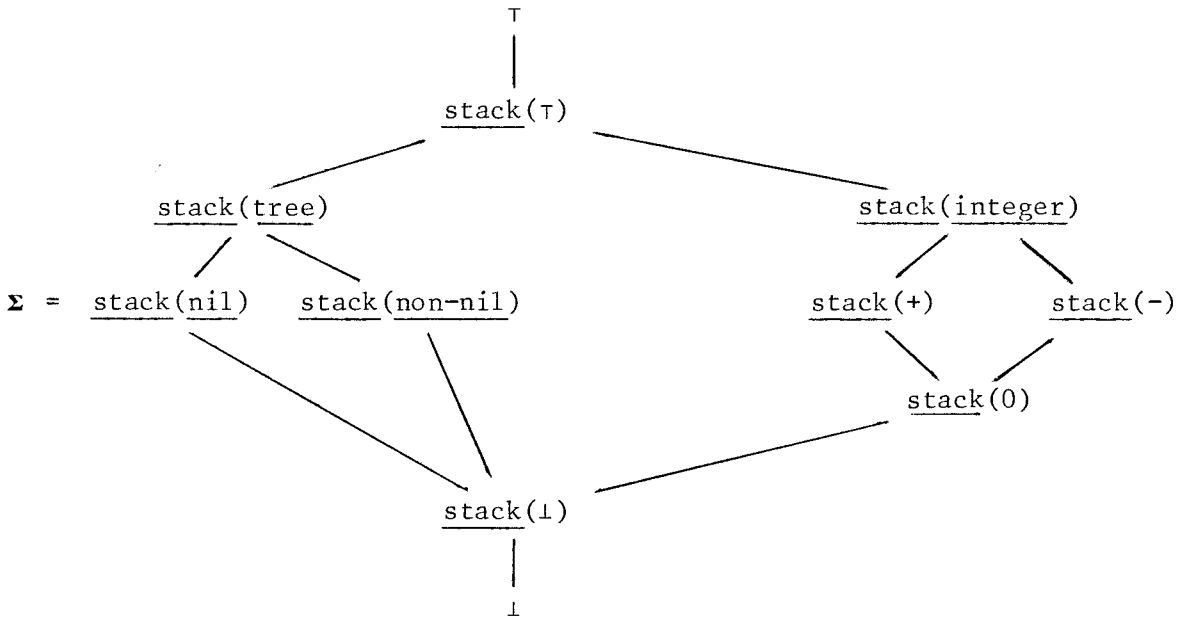
This partition of integers permits to distinguish positive from negative integers setting aside their absolute values.

The set of trees will be partitionned into either nil trees or non-nil trees :



This partition of trees is exclusive of the nodes content, of the tree shapes, etc.

The partition of the set of stacks will be thiner than the previous ones. We will distinguish between empty and non-empty stacks but a further distinction will be established between non-empty stacks depending on the nature of the stacked elements. This nature of stacked elements will be represented by the union of the abstract values of each of the stacked elements. Therefore no distinction is introduced between individual elements of the stack. A further radical simplification is introduced by considering only stacks of trees and stacks of integers :

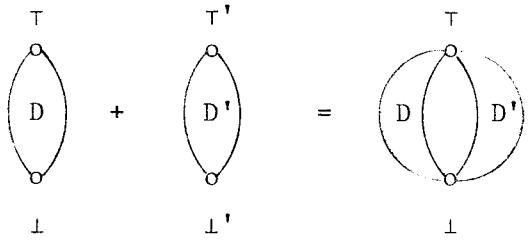


With the above scheme an empty stack is represented by the abstract value stack(⊥). A stack of positive or nul integers would be represented by stack(+). A may be empty stack containing negative integers and non-nil trees would be represented by stack(⊤), so that the content of this stack is in fact not recorded.

Finally the space **A** of abstract values for each program variable is chosen to be :

$$\mathbf{A} = \mathbf{I} + \mathbf{T} + \Sigma$$

where the plus operation between lattices is defined as follows (Scott [1971]) :



The abstract context associated with each program point of a program with n variables is a member of $\mathbf{A}^n$. We will represent an abstract context

by a sequence of couple (variable identifier ← abstract value) such as :

C_1 = <STACK ← stack(⊥), COUNT ← 0, TREE ← tree>

As before $C_1(V)$ is the value of variable "V" in context "C_1" and "$C_1(V \leftarrow a)$" is a copy of "C_1" where the value of variable "V" is replaced by the abstract value "a". For example :

C_1(TREE) = tree

C_1(TREE ← nil) = <STACK ← stack(⊥), COUNT ← 0, TREE ← nil>

We note ⊔ the union of abstract values in the lattice **A**. The operation $\sqcup_n$ is the corresponding operation on abstract contexts :

C_1 = <STACK ← stack(+), TREE ← nil>

C_2 = <STACK ← stack(-), TREE ← non-nil>

$C_1 \sqcup_2 C_2$ = <STACK ← stack(+) ⊔ stack(-), TREE ← nil ⊔ non-nil>

= <STACK ← stack(integer), TREE ← tree>

Let us now perform the abstract evaluation of a sample program which iteratively traverses a binary tree and counts its tips (Knuth[1968], Burstall[1974]) :

```
C1 -        STACK := empty; count := 0;
C2 -  loop :
C3 -        if TREE ≠ nil then
C4 -             Push TREE onto STACK;
C5 -             TREE ← left(TREE);
                 go to loop;
C6 -        else
C7 -             COUNT := COUNT+1;
C8 -             if STACK = empty then
                       go to finish;
C9 -             fi;
C10 -            Pop TREE from STACK;
C11 -            TREE := right(TREE);
                 go to loop;
            fi;
    finish :
C12 -
```

fig 8.1.1.a

Before building the corresponding system of equations we must define the abstract interpretation of primitive operations of the language.

Arithmetic operations are interpreted by the classical rules of signs.

The operations "left" and "right" are defined on non-nil trees and deliver an eventually nil tree. Therefore the operations "left(TREE)" and "right(TREE)" in a context "C" are interpreted as "$\widetilde{son}$(C(TREE))" which is defined by :

$\widetilde{son}(\underline{tree}) = \underline{tree}$
$\widetilde{son}(\underline{non\text{-}nil}) = \underline{tree}$
$\widetilde{son}(\bot) = \bot$
$\widetilde{son}(a) = \top$ for any other abstract value "a" in **A**.

The test "TREE $\neq$ nil" is defined only for trees, so that it aborts for other values. Therefore on the true path "TREE" must be a non-nil tree whereas on the false path it must be a nil tree.

Finally we have to define the abstract interpretation of stack primitives.

The assignment "STACK := empty" in context "C" delivers a context "C(STACK $\leftarrow$ $\underline{stack}(\bot)$)".

The operation "Push V onto STACK" is interpreted in context "C" by the abstract operation :

C(STACK $\leftarrow$ $\widetilde{Push}$(C(STACK), C(V)))

which is defined by :

$\widetilde{Push}(\bot, a) = \bot$
$\widetilde{Push}(\underline{stack}(a), a') = \underline{stack}(a \sqcup a')$

since the properties of the stacked elements are represented by the union of individual elements. A last rule expresses that "Push" is not defined for non-stack values :

$\widetilde{Push}(a, b) = \top$ for other $(a, b) \in \mathbf{A}^2$

The operation "Pop V onto STACK" is interpreted in a context C by the abstract operation :

$$C(V \leftarrow \widetilde{Top}(C(STACK)),\ STACK \leftarrow \widetilde{Pull}(C(STACK)))$$

defined by :

$$\widetilde{Top}(\perp) = \perp,\quad \widetilde{Pull}(\perp) = \perp$$
$$\widetilde{Top}(\underline{stack}(a)) = a,\quad \widetilde{Pull}(\underline{stack}(a)) = \underline{stack}(a)$$

The properties of the top element in the stack are the ones of the union of stacked elements since no distinction is made between individual elements. This interpretation does not take account of the length of the stack so that underflow is ignored except when "a = ⊥" which permits to report a possible program abortion. The remaining cases are covered by the rules :

$$\widetilde{Top}(a) = \tau\quad ,\ \widetilde{Pull}(a) = \tau$$

which specify that stack operations cannot be correctly performed on non-stack arguments.

The above considerations permit to associate the following system of equations to our example program (fig 8.1.1.a) :

$$\left\{\begin{array}{l}
C_1 = \langle STACK \leftarrow \underline{stack}(\perp),\ COUNT \leftarrow 0,\ TREE \leftarrow \underline{tree}\rangle \\
C_2 = C_1 \sqcup_3 C_5 \sqcup_3 C_{11} \\
C_3 = C_2(TREE \leftarrow \underline{non\text{-}nil}) \\
C_4 = C_3(STACK \leftarrow \widetilde{Push}(C_3(STACK),\ C_3(TREE))) \\
C_5 = C_4(TREE \leftarrow \widetilde{son}(C_4(TREE))) \\
C_6 = C_2(TREE \leftarrow \underline{nil}) \\
C_7 = C_6(COUNT \leftarrow (C_6(COUNT) \boxplus +)) \\
C_8 = C_7(STACK \leftarrow \underline{stack}(\perp)) \\
C_9 = C_7 \\
C_{10} = C_9(TREE \leftarrow \widetilde{Top}(C_9(STACK)),\ STACK \leftarrow \widetilde{Pull}(C_9(STACK))) \\
C_{11} = C_{10}(TREE \leftarrow \widetilde{son}(C_{10}(TREE))) \\
C_{12} = C_8
\end{array}\right.$$

fig 8.1.1.b

The least solution of the above system of equations is the limit of the finite ascending sequence of successive approximations. The initial step is given by :

C_i = <STACK ← ⊥, COUNT ← ⊥, TREE ← ⊥> i = 1 .. 12

The computations are then a bit tedious (but not difficult) so that we directly give the final result :

	STACK	COUNT	TREE
C_1	stack(⊥)	0	tree
C_2	stack(non-nil)	+	tree
C_3, C_4	stack(non-nil)	+	non-nil
C_5	stack(non-nil)	+	tree
C_6, C_7	stack(non-nil)	+	nil
C_{10}	stack(non-nil)	+	non-nil
C_{11}	stack(non-nil)	+	tree
C_8, C_{12}	stack(⊥)	+	nil

This very simple abstract evaluation of the program allows the compiler to discover program properties which turn out to be essential for code generation :

- COUNT is a positive integer
- TREE is not nil at lines 4 and 10 and therefore the operations "left" and "right" will not abort (provided that TREE is correctly initialized by a tree value which has been assumed in this interpretation).
- STACK is a stack of non-nil trees (when not empty).
- Finally, if the program terminates clearly, the final values of STACK and TREE will be respectively "empty" and "nil".

8.1.2 Compilers Must Perform an Approximate Analysis of Programs

One can make the previous interpretation more accurate by considering a set of abstract values which partition more precisely the space of

concrete program data. For example we could have represented a non-nil tree which nodes contain stacks of positive integers by the abstract value :

non-nil(stack(+))

Another refinement could be to take account of the maximal depth of trees, or the maximal length of stacks.

However these refinements imply considering an infinite space of abstract values and consequently some approximation sequences may be infinite. For example, an erroneous sequence of instructions such as :

```
...
STACK := empty ;
while ... do
     Push STACK onto STACK;
     ...
od;
...
```

would lead to the infinite abstract value :

stack(stack(stack(... stack(⊥))))

The way compilers can cope with infinite interpretations is by discovering correct approximations of the exact properties, ((4.4), they can also ask programmers for rescue (4.1.1) but this is not considered here).

8.2 *Approximation of Infinite Abstract Evaluations of Programs*

8.2.1 Structural Approximation Method

The first approximation technique we used consists in modelling the concrete data space by abstract values which cannot work up infinite approximation sequences.

Example

Suppose we want to staticly determine the set of values which each integer variable may take at each program point, so that for the non terminating program :

```
          i := 1;
C1 -
          while true do
C2 -
                i := i+1;
C3 -
          od;
C4 -
```

we would have :

$$\begin{cases} C_1 = \langle i \leftarrow \{1\} \rangle \\ C_2 = \langle i \leftarrow \{2k+1 \mid k \geq 0\} \rangle \\ C_3 = \langle i \leftarrow \{2k+3 \mid k \geq 0\} \rangle \\ C_4 = \langle i \leftarrow \emptyset \rangle \end{cases}$$

This interpretation clearly involves infinite approximation sequences. We proposed two abstractions :

- Determination of the sign (see 1) which represents an empty set by "$\perp$", a set of positive integers by "$\dot{+}$", a set of negative integers by "$\dot{-}$" and other sets by "$\dot{\pm}$".

- Constant propagation (see 5.1) which represents an empty set by "$\perp$", a set "{a}" containing a single integer by "a", and other sets by "$\top$".

- One can also imagine parity determination which represents an empty set by "$\perp$", a set of odd integers by "odd", a set of even integers by "even" and other sets by "$\top$".

8.2.2 Computational Approximation Method

A second approximation technique we will examplify now consists in modelling the concrete data space by infinitely many abstract values. Therefore this might lead to infinite approximation sequences. Yet in such a case, the limit of these sequences will be approximated in a finite number of steps using simple heuristics.

Example

A set of integers may be abstracted by its minimal and maximal members. For example the abstraction of the set {-1, 10, 5, 7} would be the interval [-1, 10].

Let us examplify interval analysis for integer variables by the trivial program :

```
                i := 1;
C1 -
     loop :
C2 -
                if i ≤ 1000 then
C3 -
                     i := i+1;
C4 -
                     go to loop;

                end;
C5 -
```

fig 8.2.2.a

We note C(i) = [a, b] the fact that a ≤ i ≤ b in context C. An obvious algorithm (see Cousot[1976]) permits to establish the following system of equations :

$$\begin{cases} C_1 = \langle i \leftarrow [1, 1] \rangle \\ C_2 = C_1 \sqcup_1 C_4 \\ C_3 = C_2(i \leftarrow C_2(i) \sqcap [-\infty, 1000]) \\ C_4 = C_3(i \leftarrow C_3(i) + [1, 1]) \\ C_5 = C_2(i \leftarrow C_2(i) \sqcap [1001, +\infty]) \end{cases}$$

fig 8.2.2.b

These equations use the union $\sqcup$ and intersection $\sqcap$ of intervals, and the operation + on intervals defined by :

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

The least solution of the above equations is given by :

$$C_2 = \langle i \leftarrow [1, 1001] \rangle$$

which permits to deduce the remaining unknowns :

$$\begin{aligned} C_3 &= C_2(i \leftarrow C_2(i) \sqcap [-\infty, 1000]) \\ &= C_2(i \leftarrow [1, 1001] \sqcap [-\infty, 1000]) \\ &= \langle i \leftarrow [\underline{\max}(1, -\infty), \underline{\min}(1001, 1000)] \rangle \\ &= \langle i \leftarrow [1, 1000] \rangle \end{aligned}$$

$$\begin{aligned} C_4 &= C_3(i \leftarrow C_3(i) + [1, 1]) \\ &= C_3(i \leftarrow [1, 1000] + [1, 1]) \\ &= \langle i \leftarrow [2, 1001] \rangle \end{aligned}$$

$$\begin{aligned} C_5 &= C_2(i \leftarrow C_2(i) \sqcap [1001, +\infty]) \\ &= C_2(i \leftarrow [1, 1001] \sqcap [1001, +\infty]) \\ &= \langle i \leftarrow [\underline{\max}(1, 1001), \underline{\min}(1001, +\infty)] \rangle \\ &= \langle i \leftarrow [1001, 1001] \rangle \end{aligned}$$

and allows the verification of the equation defining C_2 :

$$\begin{aligned} C_2 &\stackrel{?}{=} C_1 \sqcup_1 C_4 \\ \langle i \leftarrow [1, 1001] \rangle &\stackrel{?}{=} \langle i \leftarrow [1, 1] \rangle \sqcup_1 \langle i \leftarrow [2, 1001] \rangle \\ &\stackrel{?}{=} \langle i \leftarrow [1, 1] \sqcup [2, 1001] \rangle \\ &\stackrel{?}{=} \langle i \leftarrow [\underline{\min}(1, 2), \underline{\max}(1, 1001)] \rangle \\ &= \langle i \leftarrow [1, 1001] \rangle \end{aligned}$$

The problem of discovering the above solution is more difficult than a simple verification. Consider for example the approximation sequence for solving the equations :

$$\begin{cases} x = [1, 1] \\ y = x \sqcup (y + [1, 1]) \end{cases}$$

It is an infinite sequence which first terms are :

$$y = \bot, [1,1], [1,2], [1,3], [1,4], \ldots$$

and which limit 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 most simple heuristics which can be used with intervals is probably the following : if a bound of an interval is not constant take it to be infinite. Intuitively this heuristics is correct since it permits to discover at least (but not at most) all the values which a program variable may take during execution.

Applying this heuristics to the example (fig 8.2.2.a) we get :

Initialization :

$C_j = \langle i \leftarrow \bot \rangle$, $j = 1 \,..\, 5$

Step 1 :

$C_1 = \langle i \leftarrow [1, 1] \rangle$

$C_2 = \langle i \leftarrow [1, 1] \rangle$

$C_3 = \langle i \leftarrow [1, 1] \rangle$

$C_4 = \langle i \leftarrow [2, 2] \rangle$

$C_5 = \bot$

Step 2 :

$C_1 = \langle i \leftarrow [1, 1] \rangle$

$C_2 = \langle i \leftarrow [1, 2] \rangle$

$C_3 = \langle i \leftarrow [1, 2] \rangle$

$C_4 = \langle i \leftarrow [2, 3] \rangle$

$C_5 = \bot$

Applying the simple heuristics to the equation associated to the loop cutpoint that is C_2 we get :

Induction Step 3 :

$C_2 = \langle i \leftarrow [1, +\infty] \rangle$

from which we derive :

$C_1 = \langle i \leftarrow [1, 1] \rangle$

$C_3 = C_2(i \leftarrow C_2(i) \sqcap [-\infty, 1000])$

$= \langle i \leftarrow [1, +\infty] \sqcap [-\infty, 1000]) \rangle$

$= \langle i \leftarrow [1, 1000] \rangle$

$C_4 = C_3(i \leftarrow C_3(i) + [1, 1])$

$= \langle i \leftarrow [2, 1001] \rangle$

$C_5 = C_2(i \leftarrow C_2(i) \sqcap [1001, +\infty])$

$= C_2(i \leftarrow [1, +\infty] \sqcap [1001, +\infty])$

$= \langle i \leftarrow [1001, +\infty] \rangle$

Step 4 :

$C_1 = \langle i \leftarrow [1, 1] \rangle$

$C_2 = C_1 \sqcup_1 C_4 = \langle i \leftarrow [1, 1001] \rangle$

$C_3 = \langle i \leftarrow [1, +\infty] \sqcap [-\infty, 1000]) = \langle i \leftarrow [1, 1000] \rangle$

$C_4 = \langle i \leftarrow [2, 1001] \rangle$

$C_5 = C_2(i \leftarrow C_2(i) \sqcap [1001, +\infty])$

$= \langle i \leftarrow [1, 1001] \sqcap [1001, +\infty] \rangle$

$= \langle i \leftarrow [1001, 1001] \rangle$

A next iteration would prove stabilization and therefore terminates the approximation sequence.

Note that by chance we have found the exact solution, but the compiler is not aware of this fact. It simply knows that the exact solution is included in the approximated one which has been automatically discovered. (The inclusion of two contexts is defined as the conjunction of the inclusion of the abstract values of each variable in these contexts. The inclusion of two intervals $[a, b] \subseteq [c, d]$ is defined by $a \leq c \leq d \leq b$).

It is important to note that because of the undecidable problems we are faced with, the approximation of infinite evaluations is valid but fundamentally incomplete. However it should be clear that this incompleteness is accepteble to compilers which never need full knowledge of the properties of the compiled programs.

(A complete explanation of the approximation method and a proof that it correctly approximates the exact solutions is beyond the scope of this paper and may be found in Cousot[1977a]. Complementary details with a particular emphasis on interval analysis are given in Cousot[1976]).

9. CONCLUSION

Abstract interpretation offers a model for static analysis of programs which exhibits a beautiful unity in the apparent diversity of approaches. The connection between the various interpretations can be made very clear by observing that some interpretations are a refinement of others (or dually their abstraction) so that the several interpretations form a lattice which more refined element is the semantics (Cousot[1977a]).

From a theoretical point of view we hope to have shown that the results of lattice theory originally used by Scott for defining the semantics of programming languages (see references in Scott[1976]) have a wide domain of applicability which highly exceeds the study of semantics. However the present development of the theory does not offer practical methods to compute fixpoints. By comparison with mathematics and numerical analysis it seems of the utmost interest to originate research on the problem of effective generation of approximate solutions to fixpoint equations in discrete domains.

From a practical point of view we hope that the model of abstract interpretation of programs will provide a useful framework for formulating the (eventuelly existing) algorithms for static analysis of programs. By eliminating what is specific to each application one can hope to get general methods for efficient resolution of fixpoint equations. This seems to be particularily true for the numerous global data flow analysis techniques which are often designed to tackle specific problems, and could be usefully generalized. The same way the numerous methods for automatic

generation of invariants in programs do not always expose clearly their underlying principles. An understanding as the generation of approximate solution to fixpoint equations is certainly the abstraction step which is necessary to make further progress.

It is of interest to compiler writers to design a catalogue of abstract interpretations which could be used to extract those properties of programs which are commonly used by compilers. Yet it is the intuitive feeling of the authors that there is no continuum between the degree of refinement of the properties to be extracted and the cost of extraction. Hence in practice the help of the programmer seems indispensible to shunt indecidability problems. This implies designing languages which permit the specification of abstract properties of programs, in particular which allow the declaration of local indications about approximate solutions to the system of equations. The next step is to offer the ability to describe new user-defined abstract interpretations with the complementary problem of designing the compiler so that it can really take account of the new informations gathered about programs. This view is in fact a claim for enriched data type, an area where there is much current activity.

Acknowledgements

We thank Mrs F. Blanc for her beautiful and careful typing of the manuscript.

10. REFERENCES

Abian and Brown[1961].

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.

Aho and Ullman[1973].

Aho, A.V., and Ullman, J.D. *The Theory of Parsing,Translation and Compiling, Vol. II : Compiling*. Prentice-Hall, Englewood Cliffs,N.J., 1973.

Aho and Ullman[1975].

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.

Algol 68[1976].

Van Wijngaarden, A., Mailloux, B.J., Peck, J.E.L., Koster, C.H.A., Sintzoff, M., Lindsey, C.H., Meertens, L.G.L.T., and Fisker, R.G. *Revised Report on the Algorithmic Language Algol 68*. Springer-Verlag, Berlin - Heidelberg - New-york, 1976.

Allen[1971].

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

Allen and Cocke[1976].

Allen, F.E., and Cocke, J. A program data flow analysis procedure. *Comm. ACM 19*, 3 (March 1976), 137-147.

Birkhoff[1973]

Birkhoff, G., *Lattice Theory*, AMS Coll. Pub., XXV, 3rd ed., Providence, R.I., 1973.

Branquart et al.[1973].

Branquart, P., Cardinael, J.P., and Levi, J. Optimized translation process : application to ALGOL 68. Proc. Int. Comp. Symp., A. Günter et al. (Eds), North-Holland, Amsterdam, 1974, 101-107.

Burstall[1974].

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.

Cheatham and Townley[1976].

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.

Cocke[1970].

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

Cocke and Kennedy[1974].

Cocke, J., and Kennedy, K. Profitability computations on program flow graphs. IBM Research Report RC 5123, T.J. Watson Research Center, Yorktown Heights, N.Y., Nov. 1974.

Cohen and Katcoff[1976].

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

Courcelle and Nivat[1976].

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

Cousot[1976].

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].

Cousot[1977a].

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.

Cousot[1977b]

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.

De Bakker and Scott[1969].

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

Deutsch[1973].

Deutsch, L.P. An interactive program verifier. Ph.D. Th., Dept. of Computer Sc., U. of California, Berkeley, June 1973.

Dijkstra[1976].

Dijkstra, E.W. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, N.J., 1976.

Effigy[1975].

King, J.C. A new approach to program testing. Proc. Int. Conf. on Reliable Software, Los Angeles, Calif., April 1975, 228-233.

Floyd[1967].

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.

Fong et al.[1975].

Fong, A., Kam, J., and Ullman, J.D. Application of lattice algebra to loop optimization. Conf. Rec. of the 2nd ACM Symp. on Principles of Programming Languages, Palo Alto, Calif., Jan. 1975, 1-9.

Goguen et al.[1977].

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

Graham[1972].

Graham, R.M. Performance prediction. Lecture Notes, Advanced Course On Soft. Eng., Techn. U. of Munich, Germany, 1972.

Grief and Waldinger[1974].

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.

Hantler and King[1976].

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

Hecht[1975].

Hecht, M.S. *A theoretical Foundation for Global Program Improvement.* American Elsevier, 1975.

Hecht and Ullman[1973].

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.

Hehner[1976].

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.

Henderson[1975].

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

Henderson and Quarendon[1974].

Henderson, P., and Quarendon, P. Finite state testing of structured programs. Proc. 1st Int. Symp. on Programming, B. Robinet (Ed.), Lecture Notes in Computer Sci., Springer-Verlag, April 1974, 72-80.

Hoare[1969].

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

Hoare[1976].

Hoare, C.A.R. An investigation into the structure of computations. Dept. of Computer Sci., the Queen's U. of Belfast, April 1976.

Höft[1976]

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

Jensen[1965].

Jensen, J. Generation of machine code in ALGOL compilers. *BIT 5*, (1965), 235-245.

Kam and Ullman[1976].

Kam, J.B., and Ullman, J.D. Global data flow analysis and iterative algorithms. *JACM 23*, 1 (Jan. 1976), 158-171.

Karr[1976].

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

Katz and Manna[1976].

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

Kennedy[1971].

Kennedy, K. A global flow analysis algorithm. *Int. J. of Computer Math.*, 3 (Déc. 1971), 5-15.

Kennedy[1975].

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.

Kennedy[1976].

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

Kennedy and Zucconi[1977].

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.

Kildall[1973].

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.

King[1969].

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

King[1976].

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

Kleene[1952].

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

Knuth[1968].

Knuth, D.E. *The Art of Computer Programming, Vol. 1, Fundamental Algorithms*. Addison-Wesley, Reading, Mass., 1968.

Ledgard[1972].

Ledgard, H.F. A model for type checking -- With an application to Algol 60. *Comm. ACM 15*, 11 (Nov. 1972).

Mac Carthy[1963].

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

Mac Neille[1937].

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

Manna[1974].

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

Manna and Vuillemin[1972].

Manna, Z., and Vuillemin, J. Fixpoint approach to the theory of computation. *Comm. ACM 15*, 7 (July 1972), 528-536.

Morel and Renvoise[1974].

Morel, E., and Renvoise, C. Etude et réalisation d'un optimiseur global. Th. 3ième cycle, Paris VI U., June 1974.

Morris[1971].

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

Naur[1966].

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

Nivat[1974].

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

Park[1969].

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

Robert[1976].

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

Schaefer[1973].

Schaefer, M. A *Mathematical Theory of Global Program Optimization*. Prentice-Hall, Englewood Cliffs, N.J., 1973.

Schwartz[1975].

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

Scott[1970].

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.

Scott[1971].

Scott, D. The lattice of flow diagrams. In Semantics of Algorithmic Languages. E. Engeler (Ed.), Lecture Notes in Math., Vol. 188, Springer Verlag, 1971, 311-366.

Scott[1976].

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

Scott and Strachey[1971].

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.

Select[1975].

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.

Sintzoff[1972].

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.

Sintzoff[1975].

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

Sintzoff[1976].

Sintzoff, M. Iterative methods for the generation of successful programs. Unpublished notes. Presented at IFIP-WG 2.3 meeting, Saint-Pierre-de-Chartreuse, Dec. 1976.

Tarjan[1976].

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

Tarski[1955].

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

Tennenbaum[1974].

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

Ullman[1973].

Ullman, J.D. Fast algorithms for the elimination of common sub-expressions. *Acta Informatica 2*, 3 (Dec. 1973), 191-213.

Urschler[1974].

Urschler, G. Complete redundant expression elimination in flow diagrams. IBM Research Report RC 4965, T.J. Watson Research Center, Yorktown Heights, N.Y., Aug. 1974.

Waldinger and Levitt[1974].

Waldinger, R., and Levitt, K.N. Reasoning about programs. *Artificial Intelligence 5* (1974), 235-316.

Wegbreit[1974].

Wegbreit, B. The synthesis of loop predicates. *Comm. ACM 17*, 2 (Feb. 1974), 102-112.

Wegbreit[1975a].

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

Wegbreit[1975b].

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.

Yonezawa 1976 .

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