Conference Record
of the
FOURTH ACM SYMPOSIUM ON
PRINCIPLES OF PROGRAMMING LANGUAGES

Papers Presented at the Symposium
Los Angeles, California
January 17-19, 1977

Sponsored by the
ASSOCIATION FOR COMPUTING MACHINERY
SPECIAL INTEREST GROUP ON AUTOMATA AND COMPUTABILITY THEORY
SPECIAL INTEREST GROUP ON PROGRAMMING LANGUAGES
1. Introduction

A program denotes computations in some universe of objects. Abstract interpretation of programs consists in using that notation to describe computations in another universe of abstract objects, so that the results of abstract execution give some informations on the actual computations. An intuitive example (which we borrow from Sintzoff [72]) is the rule of signs. The text \(-1515 + 17\) may be understood to denote computations on the abstract universe \((+, -, \emptyset)\) where the semantics of arithmetic operators is defined by the rule of signs. The abstract execution \(-1515 + 17 \Rightarrow -(+) \times (+) \Rightarrow (-) \times (+) \Rightarrow (-)\), proves that \(-1515 + 17\) is a negative number. Abstract interpretation is concerned by a particular underlying structure of the usual universe of computations (the sign, in our example). It gives a summary of some facets of the actual executions of a program. In general this summary is simple to obtain but inaccurate (e.g. \(-1515 + 17 \Rightarrow -(+) + (+) \Rightarrow (-) + (+) \Rightarrow \emptyset\)). Despite its fundamentally incomplete results abstract interpretation allows the programmer or the compiler to answer questions which do not need full knowledge of program executions or which tolerate an imprecise answer, (e.g. partial correctness proofs of programs ignoring the termination problems, type checking, program optimizations which are not carried in the absence of certainty about their feasibility, ...). Abstract program properties are modeled by a complete semilattice, Birkhoff[61]. Elementary program constructs are locally interpreted by order preserving functions which are used to associate a system of recursive equations with a program. The program global properties are then defined as one of the extreme fixpoints of that system, Tarski[55]. The abstraction process is defined in section 6. It is shown that the program properties obtained by an abstract interpretation of a program are consistent with those obtained by a more refined interpretation of that program. In particular, an abstract interpretation may be shown to be consistent with the formal semantics of the language. Levels of abstraction are formalized by showing that consistent abstract interpretations form a lattice (section 7). Section 8 gives a constructive definition of abstract properties of programs based on constructive definitions of fixpoints. It shows that various classical algorithms such as Kildall [73], Wegbreit[75] compute program properties as limits of finite Kleene[52]'s sequences. Section 9 introduces finite fixpoint approximation methods to be used when Kleene's sequences are infinite, Cousot[76]. They are shown to be consistent with the abstraction process. Practical examples illustrate the various sections. The conclusion points out that abstract interpretation of programs is a unified approach to apparently unrelated program analysis techniques.

2. Summary

Section 3 describes the syntax and mathematical semantics of a simple flowchart language, Scott and Strachey[71]. This mathematical semantics is used in section 4 to build a more abstract model of the semantics of programs, in that it ignores the sequencing of control flow. This model is taken to be the most concrete of the abstract interpretations of programs. Section 5 gives the formal definition of the abstract interpretations of a program.

---

* Attache de Recherche au C.N.R.S., Laboratoire Associé n° 7.
** This work was supported by IRIA-SESORI under grants 73-035 and 76-160.

3. Syntax and Semantics of Programs

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

3.1 Syntax of a Program

A program is built from a set "Nodes". Each node has successor and predecessor nodes: $\neg\text{succ}$, $\neg\text{pred}$ : Nodes $\rightarrow$ 2Nodes $\neg\text{succ}$ \(\in\text{succ}(n)) \iff (n \in \neg\text{pred}(m))$

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

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

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

This section develops a simple 'mathematical semantics' of programs, in the style of Scott and Strachey[71].

- If $S$ is a set we denote $S^*$ the complete lattice obtained from $S$ by adjoining $\{\bot, \top\}$ to it, and imposing the ordering $\frac{x}{\bot} \leq x \leq \frac{\top}{\top}$ for all $x \in S$.

- The semantic domain "Values" is a complete lattice which is the sum of the lattice Bool = $\{\text{true}, \text{false}\}$ and some other primitive domains.

- Environments are used to hold the bindings of identifiers to their values:
  
  $\text{Env} = \text{Ident}^S \rightarrow \text{Values}$

  - We assume that the meaning of an expression $\text{expr} : \text{Expr}$ in the environment $e : \text{Env}$ is given by $\text{val}$.
  
    $\text{val} : \text{Expr} \rightarrow [\text{Env} \rightarrow \text{Values}]$.

    In particular the projection $\text{val} \mid \text{Bxexpr}$ of the function $\text{val}$ in domain $\text{Bxexpr}$ has the functionality:

    $\text{val} \mid \text{Bxexpr} : \text{Bxexpr} \rightarrow [\text{Env} \rightarrow \text{Booll}]$.

    - The state set "States" consists of the set of all information configurations that can occur during computations:
      
      $\text{States} = \text{States}^8 \times \text{Env}$.

    A state $(s, e : \text{States})$ consists in a control state $\text{e}(s)$ and an environment $\text{en}(s)$, such that:

      $\forall s \in \text{States}, s = \langle s(e), \text{en}(s) \rangle$.

    - We use a continuous conditional function $\text{cond}(b, e_1, e_2)$ equal to $e_1$ or $e_2$ respectively as the value of $b$ is $\text{true}$, $\text{false}$ or $\text{false}$. We also use if $b$ then $e_1$ else $e_2$ if to denote $\text{cond}(b, e_1, e_2)$.

      $\langle e \in \text{Env}, v \in \text{Values}, x \in \text{Ident} \rangle$ then $\langle e[v/x] = k, \text{cond}(y = x, v, e(y)) \rangle$.

      - The state transition function defines for each state a next state (we consider deterministic programs):

        $\text{n-state} : \text{States} \rightarrow \text{States}$

$$\text{n-state}(s) = \begin{cases} \text{let } n \in \text{end}(e)(s) \text{, then } e \in \text{en}(s) \text{ within } & \\
\text{case } n \text{ in } & \\
\text{Assignments} = & \\
\langle \text{a-succ}(n), \text{val} \mid \text{expr}(n) \mid (e / \text{id}(n)) \rangle & \\
\text{Tests} = & \\
\text{cond}(\text{val} \mid \text{test}(n) \mid (e / \text{id}(n))) & \\
\text{Junctions} = & \\
\langle \text{a-succ}(n), e \rangle & \\
\text{Exits} \Rightarrow & s \text{ exac} \end{cases}$$

      (Each partial function $f$ on a set $S$ is extended to a continuous total function on the corresponding domain $S^\top$ by $f(t) = \bot$, $f(\top) = \top$ and $f(x) = x$ if the partial function is undefined at $x$).

    - Let $\text{I-Env}$ be the bottom function on $\text{Env}$ such that:
      
      $\langle \forall e \in \text{Ident}^S, \text{I-Env}(x) = \langle \text{Values} \rangle \rangle$.

    Let $\text{I-states}$ be the subset of initial states:

    $\text{I-states} = \langle \langle \text{a-succ}(n), \text{I-Env} \rangle : m \in \text{Entries} \rangle$
- A "computation sequence" with initial state $i_0 \in I$-states is the sequence:
  $$s_n = n\text{-state}^n(i_0)$$
  for $n = 0, 1, \ldots$
  where $\tau^0$ is the identity function and $\tau^{n+1} = f \circ \tau^n$.

- The initial to final state transition function:
  $$n\text{-state}^\infty : \text{States} \rightarrow \text{States}$$
  is the minimal fixpoint of the functional:
  $$\lambda F. (n\text{-state} \circ F)$$

Therefore
  $$n\text{-state}^\infty = Y_{\text{States} \rightarrow \text{States}} (\lambda F. (n\text{-state} \circ F))$$
where $Y_f(f)$ denotes the least fixpoint of $f : D \rightarrow D$, Tarski [55].

4. Static Semantics of Programs

The constructive or operational semantics of programs defined in section 3 considers the sequence in which states occur during execution. The fundamental remark of Floyd [67] is to prove that static properties of programs it is often sufficient to consider the set of states associated with each program point.

Hence, we define the context $C_q$ at some program point $q \in \text{ArCs} \times \text{Contexts}$ of a program $P$ to be the set of all environments which may be associated to $q$ in all the possible computation sequences of $P$:

$$C_q \in \text{Contexts} = 2^{\text{Env}}$$

$$C_q = \{ e \mid (\exists i_0 = i_0 \in I\text{-states} \mid <q, e> = n\text{-state}^0(i_0)) \}$$

The context vector $C_v$ associates a context to each of the program points of a program $P$:

$$C_v \in \text{Contexts} \times \text{Contexts} = \text{Contexts}^2$$

$$\lambda q. (e, (\exists i_0 = i_0 \in I\text{-states} \mid <q, e> = n\text{-state}^0(i_0)))$$

According to the semantics of programs, the context $C_v(r)$ associated to arc $r$ is related to the contexts $C_v(q)$ at arc $q$ adjacent to $r$, $(end(q) = \text{origin}(r), \overline{q \rightarrow r})$. From the definition of the state transition function we can prove the equation:

$$C_v(r) = n\text{-context}(r, C_v)$$

where

$$n\text{-context} : \text{ArCs} \times \text{Contexts} \rightarrow \text{Contexts}$$

is defined by:

$$n\text{-context}(r, C_v) = \begin{cases} \text{case origin}(r) \in \text{Entries} \rightarrow \{ 1 \}^{\text{Env}} \\ \text{Assignments} \cup \text{Tests} \cup \text{Junctions} \rightarrow \bigcup_{e \in C_v(q)} e \circ \text{env-on}(r)(n\text{-state}(<q, e>)) \\ e \circ \text{env-on}(r)(n\text{-state}(<q, e>)) \end{cases}$$

$$\text{case stmt}(x, y)$$

(We define $\text{env-on} : \text{ArCs} \rightarrow [\text{States} \rightarrow 2^{\text{Env}}]$ to be $\lambda r. (\lambda s. \text{cond}(r = \text{ex}(s), (\text{env}(s), \emptyset)))$.

Since the equation $C_v(r) = n\text{-context}(r, C_v)$ must be valid for each arc, $C_v$ is a solution to the system of "forward" equations:

$$C_v = F\text{-cont}(C_v)$$

where

$$F\text{-cont} : \text{Contexts} \times \text{Contexts}$$

is defined by:

$$F\text{-cont}(C_v) = \lambda q. n\text{-context}(r, C_v)$$

$\text{Contexts}$ is a complete lattice with union $\vee$ such that $C_v \cup C_v = \lambda r. (C_v(r) \cup C_v(r))$. $F\text{-cont}$ is order preserving for the ordering $\leq$ of $\text{Contexts}$ which is defined by:

$$C_v \leq C_v \iff (\forall r \in \text{ArCs}, C_v(r) \leq C_v(r))$$

Hence it is known that $F\text{-cont}$ has fixpoints, Tarski [55]. However, it is trivial to exhibit examples which show that these fixpoints are not always unique. Fortunately, it can be shown that $C_v$ is included in any solution $G$ to the system of equations $X = F\text{-cont}(X)$, $(C_v \leq G)$. Tarski [55] shows that this property uniquely determines $C_v$ as the least fixpoint of $F\text{-cont}$. Thus $C_v$ can be equivalently defined by:

$$D_1 : C_v = \lambda q. (e, (\exists i_0 \in I\text{-states} \mid <q, e> = n\text{-state}^0(i_0)))$$

$$D_2 : C_v = Y_{\text{Contexts} \times \text{Contexts}} (F\text{-cont})$$

The concrete context vector $C_v$ is such that for any program point $q \in \text{ArCs} \times \text{Contexts}$ of the program $P$,

(a) $C_v(q)$ contains at least the environments $e$ which may be associated to $q$ during any execution of $P$:

$$\{ q, e \mid (\exists i_0 \in I\text{-states} \mid <q, e> = n\text{-state}^0(i_0)) \}$$

(b) $C_v(q)$ contains only the environments $e$ which may be associated to $q$ during an execution of $P$:

$$\{ q, e \mid (\exists i_0 \in I\text{-states} \mid <q, e> = n\text{-state}^0(i_0)) \}$$

$C_v$ is merely a static summary of the possible executions of the program. However, our definitions $D_1$ or $D_2$ of $C_v$ cannot be utilized at compile time since the computation of $C_v$ consists in fact in running the program (for all the possible input data). In practice compilers may consider states which can never occur during program execution (e.g. some compilers consider that any program may always perform a division by zero although this is not the case for most programs). Hence compilers may use "abstract" contexts satisfying (a) but not necessarily (b), which therefore correctly approximate the concrete contexts we considered until now.

5. Abstract Interpretation of Programs

5.1 Formal Definition

An abstract interpretation $I$ of a program $P$ is a tuple

$$I = \langle A\text{-Cont}, \leq, \leq, \circ, \text{Int} \rangle$$

where the set of abstract contexts is a complete $o$-semilattice with ordering $\leq$ $(x \leq y) \iff (x \leq y)$. This implies that $A\text{-Cont}$ has a supremum $\vee$. We suppose also $A\text{-Cont}$ to have an infimum $\wedge$. 
This implies that \( A\text{-}Cont \) is in fact a complete lattice, but we need only one of the two join and meet operations. The set of context vectors is defined by 
\( A\text{-}Cont = \text{Arcs}^0 \to A\text{-}Cont \).

Whatever \((Cv', Cv'') \in A\text{-}Cont^2\) may be, we define:
\[
Cv' \triangleright Cv'' = \lambda r. Cv'(r) \circ Cv''(r) \\
Cv' \trianglerighteq Cv'' = \{ \forall r \in \text{Arcs}^0, Cv'(r) \leq Cv''(r) \}
\]

\( \top = \lambda r. \top \) and \( \bot = \lambda r. \bot \)

\( \langle A\text{-}Cont, \triangleright, \trianglerighteq, \top, \bot \rangle \) can be shown to be a complete lattice. The function:
\[
\text{Int} : \text{Arcs}^0 \times A\text{-}Cont \to A\text{-}Cont
\]
defines the interpretation of basic instructions. If \( \{C(q)| q \in a\text{-}prod(n)\} \) is the set of input contexts of node \( n \), then the output context on exit arc \( r \) of \( n \) \((r \in a\text{-}succ(n))\) is equal to \( \text{Int}(r, C) \).

\( \text{Int} \) is supposed to be order-preserving:

\[
\forall a \in \text{Arcs}, \forall (Cv', Cv'') \in A\text{-}Cont^2, \quad (Cv' \trianglerighteq Cv'') \Rightarrow (\text{Int}(a, Cv') \leq \text{Int}(a, Cv''))
\]

The local interpretation of elementary program cons-

\[\text{Examples:}\]
Kildall[73] uses \((n, \rightarrow, \top)\), Wegbreit[75] uses \((v, \rightarrow, \top)\). Tenenbaum[74] uses both \((u, \rightarrow, \top)\) and \((n, \rightarrow, \top)\).

\[\text{5.3 Examples}\]
\[\text{5.3.1 Static Semantics of Programs}\]

The static semantics of programs we defined in section 5 is an abstract interpretation:
The determination of available expressions, back-dominators, intervals, ... requires a forward system of equations. Some global flow problems, notably the live variables and very busy expressions require propagating information backward through the program graph, they are examples of backward systems of equations.

6.5 Remarks

Our formal definition of abstract interpretations has the completeness property since the model ensures the existence of a particular solution to the system of equations and therefore defines at least some global property of the program. It must also have the consistency property, that is define only correct properties of programs.

One can distinguish between syntactic and semantic abstract interpretations of a program. Syntactic interpretations are proved to be correct by reference to the program syntax (e.g., the algorithm for finding available expressions is justified by reasoning on paths of the program graph). By contrast semantic abstract interpretations must be proved to be consistent with the formal semantics of the language (e.g., constant propagation).

6. Consistent Abstract Interpretations

An "abstract" interpretation \( \bar{T} = (\bar{\text{A-Cont}}, \bar{\gamma}, \bar{x}, \bar{T}, \bar{1}, \bar{\text{Int}}) \) of a program is consistent with a "concrete" interpretation \( T = (\text{C-Cont}, \gamma, x, \bar{T}, 1, \text{Int}) \) if the context vector \( \bar{\text{CV}} \) resulting from \( \bar{T} \) is a correct approximation of the particular context \( \text{CV} \) resulting from the more refined interpretation \( T \). This may be rigorously defined by establishing a correspondence (\( \alpha : \) abstraction) between concrete and abstract context vectors, and inversely (\( \bar{\gamma} : \) concretization), and requiring:

6.0 \( (\bar{\gamma}(\bar{x}) \preceq \bar{\gamma}(\text{CV})) \) and \( (\alpha(\text{CV}) \preceq \text{CV}) \)

In words the abstract context vector must at least contain the concrete one, (but not only the concrete one).

If \( f : D \to D' \) we note \( \bar{D} = \text{Arcs}^\circ \to D \) and \( \bar{D}' = \text{Arcs}^\circ \to D' \) and \( \bar{f} : \bar{D} \to \bar{D}' = \lambda \lambda . (\lambda r . f(d(r))) \).

We will suppose \( \alpha \) and \( \bar{\gamma} \) to satisfy the following hypothesis:

6.1 \( \alpha : \text{C-Cont} \to \text{A-Cont} \), \( \bar{\gamma} : \text{A-Cont} \to \text{C-Cont} \)

6.2 \( \alpha \) and \( \bar{\gamma} \) are order-preserving.

6.3 \( \forall x \in \text{A-Cont}, \bar{x} = \alpha(\bar{\gamma}(x)) \)

6.4 \( \forall x \in \text{C-Cont}, x \preceq \gamma(\alpha(x)) \)

Intuitively, hypothesis 6.2 is necessary because context inclusion (that is property comparison) must be preserved by the abstraction or concretization process. 6.3 requires that concretization introduces no loss of information. It implies that \( \alpha \) is surjective and \( \bar{\gamma} \) is injective. 6.4 introduces the idea of approximation: the abstraction \( \alpha(C) \) of a concrete context \( C \) may introduce some loss of information so that when concretizing again \( \gamma(\alpha(C)) \) we may get a larger context \( \gamma(\alpha(C)) \supset C \). Note that it is easy to prove properties corresponding to 6.1-6.4 for \( \alpha \) and \( \bar{\gamma} \).

Instead of the global hypothesis 6.0 we will use the following local hypothesis on the concrete and abstract interpretations of primitive language constructs:

\( (\forall a, \bar{x} \in \text{Arcs} \times \text{A-Cont}, \gamma(\text{Int}(a, \bar{x})) \preceq \text{Int}(a, \bar{\gamma}(x))) \)

6.5 \( (\forall a, x \in \text{Arcs} \times \text{C-Cont}, \text{Int}(a, \alpha(x)) \preceq \alpha(\text{Int}(a, x))) \)

These two hypothesis are in fact equivalent (lemma 1.2 in appendix 12). The following schema illustrates 6.5, i.e. the idea of abstract simulation of concrete computations:

\( \bar{\gamma} \)

\( \alpha \)

\( \gamma \)

\( \text{Int}(a, \bar{C}) \)

\( \bar{\gamma} \)

\( \gamma \)

\( \alpha \)

\( \text{Int}(a, C) \)

Suppose we want to compute the concrete output context \( C_0 \) (associated with arc \( a \)) resulting from concrete input contexts \( \bar{C} \) : \( C_0 = \text{Int}(a, \bar{C}) \). We can as well approximate this computation in the abstract universe, and get \( \bar{C}_0 = \gamma(\text{Int}(a, \alpha(\bar{C}))) \). 6.5 requires \( \bar{C}_0 \) to contain at least \( \bar{C}_0 \), that is \( \bar{C}_0 \preceq \bar{C}_0 \). On the contrary we do not require \( \bar{C}_0 \) to contain at most \( \bar{C}_0 \), that is \( \bar{C}_0 \preceq \bar{C}_0 \) is not compulsory.

We will say that \( \bar{T} \) is a refinement of \( T \), or that \( \bar{T} \) is an abstraction of \( T \), denoted \( \bar{T} \preceq (\gamma, \alpha) \bar{T} \), if and only if there exist \( \alpha \) and \( \bar{\gamma} \) satisfying hypothesis 6.1 to 6.3.

Note that \( \bar{T} \preceq (\gamma, \alpha) \bar{T} \) implies a local consistency of the interpretations \( \bar{T} \) and \( T \), at the level of primitive language constructs (6.5). Theorems 7 and 7 of Appendix 12 then prove 6.0 which defines the global consistency of \( T \) and \( \bar{T} \) at the program level.

In particular if we take

\( I_{\text{SS}} = \{\text{Contexts}, u, \preceq, \text{Env}, \emptyset, \alpha, \text{p-context}\} \)

any abstract interpretation \( \bar{T} \) of \( P \), consistent with

\( I_{\text{SS}} (I_{\text{SS}} \preceq (\alpha, \gamma) \bar{T}) \)

is consistent with the semantics of \( P \), which implies:

\( \forall x \in \text{Arcs}, \bar{\gamma} \bar{\text{CV}} \) be the result of \( \bar{T} \),

\( \exists n < 0, \exists i_s \in \text{I-states} \mid < q_s, e_s > \equiv \text{n-scale}^p(i_s) \implies (e \in \gamma(\bar{x}(q_s))) \)

As previously noticed, the abstract interpretations will not in general be powerful enough to establish the reciprocal.

Example: Deductive Semantics of Programs

Contexts will be predicates such as \( P(x_1, \ldots, x_n) \) \( \in \text{Pred} \) over the program variables \( (x_1, \ldots, x_n) \) \( \in \text{Ident}^n \) which are the free variables in the predicate. The abstract interpretation is then:

\( I_{\text{DS}} = \{\text{Pred}, \text{or}, \text{eq}, \text{true}, \text{false}, \text{n-pred}\} \)
where $n$-pred defines Floyd's strongest postcondition:

$$n\text{-pred}(r, P) = \begin{cases} \text{let}(s \text{ be origin}(r), \langle p \text{ be } a\text{-pred(origin}(r)) \rangle \text{ within case } \eta \text{ in Entries} \Rightarrow (\forall x \in \text{Ident}, x = \text{Values}) \end{cases}$$

The relation $\equiv$ on abstract interpretations defined by:

$$[I \equiv I'] \iff [(I \leq I') \text{ and } (I' \leq I)]$$

is an equivalence relation. We have:

$$[I \equiv (\beta)I'] \iff \{\beta \text{ is an isomorphism between the algebras } I \text{ and } I'\}$$
A further abstraction may be:

\[ \alpha(a, b) = \begin{cases} +1 & \text{if } a = b \\ \alpha & \text{if } a > b \\ \beta & \text{if } a < b \end{cases} \]

The abstract contexts are then:

\[ I_C = I_R = \ldots, I = \ldots, I_{CS} = \ldots, I_{CP} = \ldots, I_{RS} = \ldots \]

This interpretation may be abstracted by two non-comparable abstractions:

\[ I_{CS} = \ldots, I_{CP} = \ldots, I_{RS} = \ldots \]

8. Abstract Evaluation of Programs

The system of equations:

\[ CV := \text{Int}(CV) \]

resulting from an interpretation \( I = \{a \rightarrow \text{Cont.} \leq, \leq, \tau, i, \text{Int} \} \) of a program \( P \) may be solved by "elimination" methods (e.g. Tarjan[75]). Otherwise, one can use an "iterative" algorithm which computes Kleene's sequence (L4 of Appendix 12):

\[ CV := (C := I; \text{until } C = \text{Int}(C) \text{ do } C := \text{Int}(C) \text{ repeat }) C \]

8.1 Correctness

If \( \text{Int} \) is supposed to be a complete morphism (i.e. infinitely distributive over \( \ldots \)) then \( CV \) is the least fixpoint of \( \text{Int} \) (e.g. Kildall[75]), since in a semi-lattice of finite length, any distributive function is a complete morphism. Under the weaker assumption that \( \text{Int} \) is continuous, the limit \( CV \) of Kleene's sequence can also be shown to be the least fixpoint of \( \text{Int} \) (e.g. Wegbreit[75], since in a well-founded semi-lattice, any isotone function is continuous). Finally, if \( \text{Int} \) is only supposed to be isotone, \( CV \) is an approximation (\( \varphi \)) of the least fixpoint (e.g. Knu and Ullman[75]).

8.2 Termination

The abstract evaluation terminates if Kleene's sequence is finite. This may be the case because \( A \rightarrow \text{Cont.} \) is finite (e.g. type checking in ALGOL 68, Naur[65]), or a finite subset only is to be considered for any particular program (e.g. type checking in ALGOL 68), or \( A \rightarrow \text{Cont.} \) may be of finite length \( m \) (the length of any strictly increasing chain is bounded by \( m \), Kildall[73], Wegbreit[75]) or \( A \rightarrow \text{Cont.} \) may satisfy the ascending chain condition (every strictly increasing chain is finite, although not bounded). A lattice may have infinite chains, although \( \text{Int} \) is chosen so that Kleene's sequences are finite. Finally an infinite Kleene's sequence may be arbitrarily truncated (to get a lower bound of its limit), some induction principle (Sintzoff [75]) or heuristics (Katz and Manual[76]) may be used to pass to the limit, or approximate it, (Cousot[76]).

8.3 Efficiency

In practice efficient versions of the Kleene's sequence are used. These consist in a symbolic execution of the program which propagates information along paths of the program until stabilization. A specification of order of information propagation may lead to optimal algorithms for specific applications (references in Tarjan[76]).
8.4 Example: Performance Analysis of Programs

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.

Let $\text{A-Cont}$ be the lattice $\mathbb{R}^+$ of positive real numbers augmented by the upper bound $\infty$, with natural ordering $\leq$. The abstract interpretation:

$$I_p = \langle \mathbb{R}^+, \max, \leq, 0, \infty, \text{Kir} \rangle$$

may be used to derive the mean values of the counters using Kirchhoff's law of conservation of flow:

$$\text{Kir}(r, \text{Cv}) =$$

\[
\begin{align*}
\text{let } n & \text{ be origin}(r) \text{ within } \\
\text{case } n \text{ in } & \\
\text{Entries} & \Rightarrow 1 \{\text{unique entry node}\} \\
\text{Junctions} \cup \text{Assignments} & \Rightarrow \prod_{p \in \text{pred}(n)} \text{Cv}(p) \\
\text{Tests} & \Rightarrow \\
\text{case } r \text{ in } & \\
\{a \rightarrow \text{true}(n)\} & \Rightarrow \text{Cv}(a \rightarrow \text{pred}(n)) \times \text{Prob}(\text{test}(n) = \text{true})
\end{align*}
\]

9. Fixpoints Approximation Methods

When the extreme fixpoints of the system of equations established for an abstract interpretation $I$ of a program $P$ cannot be computed in finitely many steps, they can be approximated. A more abstract interpretation $I (I \leq I')$ may be used for that purpose (e.g., Temenbaum[74]). It is often better to make approximations in $I$, for example by "accelerating the convergence" of Kleene's sequences.

9.1 Finite Iterative and Increasing Approximation of the Least Fixpoint Starting from a Lower Bound

Let $I = \langle \text{A-Cont}, \text{a}, \leq, 1, \tau, \text{Int} \rangle$ be an interpretation of $P$. When the least fixpoint $\text{Cv}$ of $\text{Int}$ is unreachable, we look for an upper bound $\text{Ub}$ of $\text{Cv}$, since according to the correctness requirement $\text{Ub} \leq \gamma(\text{Ub})$.

9.1.1 Increasing Approximation Sequence
9.1.2 Generalization of Kleene's Ascending Sequence

When $\Lambda$-Cont satisfies the ascending chain condition one can choose $\Lambda\text{-}\text{Int}$ to be $\text{Int}$ and therefore the approximation sequence generalizes Kleene's sequence and the related methods.

9.1.3 Widening in Increasing Approximation sequences

The definition of the approximate interpretation as before, we define:

9.1.3.5 $\Lambda\text{-}\text{Int} = \lambda q . \Lambda\text{-}\text{Int}(q, Cv)$

Now we have to show that this definition of $\Lambda\text{-}\text{Int}$ satisfies the requirements 9.1.1.2 and 9.1.1.7.

Let us consider a sequence $S_0 = \tilde{1}, \ldots, S_n = \Lambda\text{-}\text{Int}(S_p), \ldots$. We show that this sequence is increasing that is to say:

9.1.3.6 $S_n \approx \Lambda\text{-}\text{Int}(S_n) \land n \geq 0$. 

Let us note \([a, b]\) where \(a \leq x \leq b\) the predicate \(a \leq x \leq b\). The system of equations corresponding to the example is:

\[
\begin{align*}
(0) & \quad C_0 = [], \\
(1) & \quad C_1 = [1, 1], \\
(2) & \quad C_2 = C_1 \cup C_4, \\
(3) & \quad C_3 = C_2 \cap [-\infty, 100], \\
(4) & \quad C_4 = C_3 \ast [1, 1], \\
(5) & \quad C_5 = C_2 \cap [101, +\infty).
\end{align*}
\]

Assignment statements are treated using an interval arithmetic (e.g. \([1, j] + [k, \ell] = [i+k, j+\ell]\) naturally extended to include the case of the empty interval). Similarly tests are treated using an "interval logic." Since there exist infinite Kleene's sequences (e.g. \(i, j \in [0, 0] < [0, 1] < \ldots < [0, +\infty]\) for the program \(x := 0; \text{while} \ x \neq -1 \text{do} x := x + 1\), we must use an approximation sequence. Hence the results will be somewhat inaccurate but runtime subscript tests may be inserted in the absence of certainty.

Let us define the widening \(\forall\) of intervals by:

\[
- [i, j] \forall [k, \ell] = \begin{cases} 
  \text{null} & \text{if } k < i \text{ then } -\infty \text{ else } i, \\
  \text{if } k > j \text{ then } +\infty \text{ else } j.
\end{cases}
\]

\(\forall\) satisfies the requirements of 9.1.3. According to 9.1.3.4 the system of equations is modified by:

\[
(2) \quad C_2 = C_2 \forall (C_1 \cup C_4).
\]

The corresponding approximation sequence is:

\[
\begin{align*}
* & \quad C_1 = [1, 1] \\
* & \quad C_2 = C_2 \forall (C_1 \cup C_4) \\
  & \quad = [1, 1] \forall ([1, 1] \cup [1], 1) \\
  & \quad = [1, 1] \forall [1, 1] \\
  & \quad = [1, 1] \\
* & \quad C_3 = C_2 \cap [-\infty, 100] \\
  & \quad = [1, 1] \cap [-\infty, 100] \\
  & \quad = [1, 1] \\
* & \quad C_4 = C_3 \ast [1, 1] \\
  & \quad = [1, 1] + [1, 1] \\
  & \quad = [2, 2] \\
* & \quad C_2 = C_2 \forall (C_1 \cup C_4) \\
  & \quad = [1, 1] \forall ([1, 1] \cup [2, 2]) \\
  & \quad = [1, 1] \forall [1, 2] \\
* & \quad C_3 = C_2 \cap [-\infty, 100] \\
  & \quad = [1, +\infty] \cap [-\infty, 100] \\
  & \quad = [1, 100] \\
* & \quad C_4 = C_3 \ast [1, 1] \\
  & \quad = [1, 100] \ast [1, 1] \\
  & \quad = [1, 100] \ast [1, 1] \\
  & \quad = [1, 100] \\
* & \quad C_5 = C_2 \cap [101, +\infty] \\
  & \quad = [1, +\infty] \cap [101, +\infty] \\
  & \quad = [101, +\infty]
\end{align*}
\]

The final context on each arc is marked by a star * . Note that the results are approximate ones, (e.g. C5).

In this example the widening is a very rough operation which introduces a great loss of information. However it can be seen in the trace that tests behave like filters. Furthermore, for PASCAL like languages, one can first use the bounds given in the declaration of \(x\) before widening to infinite limits.

### 9.3 Finite Iterative and Decreasing Approximation of the Least Fixpoint Starting from an Upper Bound

The ascending approximation sequence leads to an upper bound \(S = \text{Int}^n(S)\) of the least fixpoint \(C^*\) of \(\text{Int}\). Moreover \(\text{Int}^n(S) \leq S\). Since \(\text{Int}\) is order preserving, this implies that:

\[
S \geq \text{Int}^n(S) \geq \ldots \geq \text{Int}^1(S) \geq S.
\]

If \(S_m\) is not a fixpoint of \(\text{Int}\) and the above descending sequence is finite (e.g. the lattice \(A\)-Cont satisfies the descending chain condition) its limit is a better approximation of \(C^*\) than \(S_m\). When the sequence is infinite or slowly converging, one can among other solutions approximate its limit.

#### 9.3.1 Decreasing Approximation Sequence

At step \(n\) in the descending sequence, we have:

\[
\text{Int}^{n+1}(S) \geq \text{Int}^n(S) \geq C^*
\]

In order to accelerate the convergence, we should for the next step find an approximation \(D\) such that \(\text{Int}^{n+1}(S) \geq D \geq C^*\). But not knowing \(C^*\), this characterization is very weak since \(D\) could be chosen incorrectly that is to say less than \(C^*\) or non comparable with \(C^*\). The fact that \(C^*\) is the greatest lower bound of the set of \(X \in A\)-Cont such that \(\text{Int}(X) \nless X\) gives a correctness criterion for the choice of \(D\) when \(C^*\) is unknown, we must have:

\[
\text{Int}^{n+1}(S) \geq D \geq \text{Int}^n(D)
\]

On the contrary to 9.1.1, this characterization does not provide an efficient construction of \(D\).

#### 9.3.2 Truncated Decreasing Sequence

In front of these difficulties we will enforce convergence by choosing \(D\) such that:

\[
\exists n \geq 0\; |\; \text{Int}(S_n) \geq D \geq \text{Int}^{n+1}(S)
\]
(However, we will not artificially truncate the decreasing sequence by imposing an arbitrary upper bound on $n$).

Let $\overline{D\text{-}\text{Int}} : A\text{-Cont} \to A\text{-Cont}$ be such that:

9.3.2.1 $\{\forall c \in A\text{-Cont} : (c \succeq \overline{\text{Int}}(c)) \implies (c \succeq \overline{D\text{-}\text{Int}}(c))\}$

9.3.2.2 If $\forall c \in A\text{-Cont}$, every infinite sequence $c, \overline{D\text{-}\text{Int}}(c), \ldots, \overline{D\text{-}\text{Int}}(c)_n, \ldots$ is not strictly decreasing.

The truncated decreasing sequence $S'_0, S'_1, \ldots, S'_n, \ldots$ is recursively defined by:

9.3.2.3 $S'_0 = S_m$

\[
S'_{n+1} = \begin{cases} S'_n & (S'_n \neq \overline{\text{Int}}(S'_n)) \text{ and } (S'_n \neq \overline{D\text{-}\text{Int}}(S'_n)) \\ \overline{D\text{-}\text{Int}}(S'_n) & \text{else} \end{cases}
\]

Let us now prove that the truncated decreasing sequence is a finite strictly decreasing chain which terms are greater than $\overline{\text{CV}}$ the least fixpoint of $\overline{\text{Int}}$.

Let $p$ be the least natural number (eventually infinite) such that $S'_p = S'_{p+1}$. Trivially from 9.1.1:

\[
S'_0 = S_m \succeq \overline{\text{Int}}(S'_0) \succeq \overline{\text{CV}}
\]

If $p > 0$ then $S'_0 \neq \overline{\text{Int}}(S'_0)$, therefore $S'_0 \succeq \overline{\text{Int}}(S'_1)$. Then applying 9.1.2.1 we have:

\[
S'_0 \succeq \overline{D\text{-Int}}(S'_1) = S'_1 \succeq \overline{\text{Int}}(S'_1) \succeq \overline{\text{CV}}
\]

But 9.3.2.3 implies $S'_0 \neq \overline{D\text{-Int}}(S'_0)$, hence:

\[
S'_0 > S'_1 \succeq \overline{\text{Int}}(S'_0) \succeq \overline{\text{CV}}
\]

For the induction step, let us suppose that for $k < p$, we have:

\[
S'_{k+1} \succeq \overline{\text{Int}}(S'_{k+1}) \succeq \overline{\text{CV}}
\]

Since $\overline{\text{Int}}$ is order preserving we have:

\[
\overline{\text{Int}}(S'_{k+1}) \succeq \overline{\text{Int}}(S'_{k+1}) ^* \succeq \overline{\text{Int}}(S'_{k+1}) \succeq \overline{\text{CV}}
\]

By transitivity $S'_k \succeq \overline{\text{Int}}(S'_k)$ and since 9.3.2.3 implies $S'_k \neq \overline{\text{Int}}(S'_k)$ we have from 9.3.2.1:

\[
S'_k \succeq \overline{D\text{-Int}}(S'_k) = S'_{k+1} \succeq \overline{\text{Int}}(S'_k)
\]

Since 9.3.2.3 implies $S'_k \neq \overline{D\text{-Int}}(S'_k)$ we have:

\[
S'_k \succeq S'_{k+1} \succeq \overline{\text{Int}}(S'_k) \succeq \overline{\text{CV}}
\]

By recurrence on $k$ the result is true for $k \leq p$.

Moreover 9.3.2.2 implies that $p$ is finite. Q.E.D.

9.3.4 Narrowing in Truncated Decreasing Sequences

By analogy with 9.1.3 we define now the narrowing operation in order to build a possible construction of $\overline{D\text{-Int}}$ by local modifications to $\overline{\text{Int}}$:

9.3.4.1 $\Delta : A\text{-Cont} \times A\text{-Cont} \to A\text{-Cont}$

9.3.4.2 $\overline{\Delta}(c, c') \in A\text{-Cont}$,

\[
(c \succeq c') \implies (c \succeq c \Delta c')
\]

9.3.4.3 Every infinite sequence $s_0, s_1, s_2, \ldots$ of the form $s_0 = C_0, s_1 = s_0 \Delta C_1, \ldots, s_n = s_{n-1} \Delta C_n, \ldots$ for arbitrary abstract contexts $C_0, C_1, \ldots, C_n, \ldots$ is not strictly decreasing.

The approximated interpretation $\overline{\text{D-int}} : \text{Arcs}^\times \times A\text{-Cont} \to A\text{-Cont}$ is defined by:

9.3.4.4 $\overline{\text{D-int}}(q, C) . \overline{\Delta}(q, C, \overline{\text{Int}}(q, C), \overline{\text{Int}}(q, C))$

and $\overline{\text{D-int}} = \lambda (q, C) . \overline{\Delta}(q, C, \overline{\text{Int}}(q, C), \overline{\text{Int}}(q, C))$.

This definition of $\overline{\text{D-int}}$ trivially satisfies the requirement 9.3.2.1 since $\overline{\text{Int}}(q, C)$ implies $\overline{\text{Int}}(q, C) \succeq \overline{\text{Int}}(q, C)$, $q \in \text{Arcs}$. If $q \in \text{W-arcs}$ then 9.3.4.2 implies that $\overline{\text{Int}}(q, C) \succeq \overline{\text{Int}}(q, C) \succeq \overline{\Delta}(q, C, \overline{\text{Int}}(q, C), \overline{\text{Int}}(q, C))$. Otherwise, if $q \notin \text{W-arcs}$ $\overline{\text{Int}}(q, C) \succeq \overline{\Delta}(q, C, \overline{\text{Int}}(q, C), \overline{\text{Int}}(q, C))$. Hence $\overline{\text{Int}}(q, C) \succeq \overline{\Delta}(q, C)$.

The proof of termination (requirement 9.3.2.2) is very similar to the one outlined for $A\text{-Cont}$ in section 9.1.3.

9.4 Example: Bounded Integer Variables

Let us come back to example 9.2. The system of equations was:

1. $C_1 = [1, 1]$
2. $C_2 = C_1 \cup C_4$
3. $C_3 = C_2 \cap [-\infty, 100]$
4. $C_4 = C_3 + [1, 1]$
5. $C_5 = C_2 \cap [101, +\infty]$

The ascending approximation sequence led to the approximate solution:

\[
\begin{align*}
C_1 &= [1, 1] \\
C_2 &= [1, +\infty] \\
C_3 &= [1, 100] \\
C_4 &= [2, 101] \\
C_5 &= [101, +\infty]
\end{align*}
\]

Let us define the narrowing $\Delta$ of intervals by:

- $[-, -]$ is the null element of $\Delta$.
- $[i, j] \Delta [k, \ell] = (\text{if } i = - \implies \text{then } k \text{ else } \min(i, k), \ell)$
- $[i, j] \Delta +\infty = [i, +\infty]$
Thus narrowing just discards infinite bounds and makes no improvement on finite bounds, it satisfies
the requirements of 9.3.4. According to 9.3.4.4 the system of equations is modified by:

\[ C_2 = C_2 \land (C_1 \lor C4) \]

The descending approximation sequence is:

\[ C_2 = C_2 \land (C_1 \lor C4) \]
\[ = [1, +\infty) \land ([1, 1] \lor [2, 10]) \]
\[ = [1, +\infty) \land [1, 10] \]
\* \[ C_2 = [1, 10] \]
\[ C_3 = C_2 \cap [-\infty, 100] \]
\* \[ C_3 = [1, 101] \cap [-\infty, 100] = [1, 100] \]
stop on that path.
\[ C_5 = C_2 \cap [101, +\infty] \]
\* \[ C_5 = [1, 101] \cap [101, +\infty] = [101, 101] \]
exit.

On that example the approximate solution has been improved so that the least fixpoint is reached but this is not the case in general.

9.5 Dual Approximation Methods

The lattice $A_{cont}$ may be partitioned as follows:

\[ \begin{array}{c}
X \subseteq \overline{\text{Int}}(X) \\
\text{non comparable}
\end{array} \]

\[ X \supseteq \overline{\text{Int}}(X) \]
\[ X = \overline{\text{Int}}(X) \]
\[ \sim \]
\[ AKS \]
\[ \ell_{fp} \]
\[ gfp \]
\[ DKS \]

$\ell_{fp}$ and $gfp$ are the least and greatest fixpoints of $\overline{\text{Int}}$. The ascending (AKS) and descending (DKS)
Kleene's sequences converge toward $\ell_{fp}$ and $gfp$ respectively. These limits are reached when $\text{Int}$ is continuous. When AKS is infinite we have proposed to use an ascending approximation sequence (AAS) to approximate $\ell_{fp}$. Its limit may be some fixpoint $\ell_{fp}$, or some $S_m$ such that $S_m \supset \overline{\text{Int}}(S_m)$ and $S_m \supset \ell_{fp}$.

When $X \supseteq Y$ we have noted $X \longrightarrow Y$.
The truncated descending sequence TDS is fundamentally different from AAS, since it ensures that the successive approximations starting from $S_m$ remain in the partition $\{X | X \supseteq \overline{\text{Int}}(X)\}$, so that their limit $S'$ is greater than $\ell_{fp}$:

\[ \overline{\text{Int}}(S_m) \]
\[ \sim \]
\[ AKS \]
\[ \ell_{fp} \]
\[ gfp \]
\[ DKS \]

It is clear that the ascending approximation sequence AAS when starting from $I$ leads to an upper
bound of the least fixpoint $\ell_{fp}$ of $\overline{\text{Int}}$, and the truncated descending sequence TDS when starting from $I$ leads to an upper bound of the greatest fixpoint $gfp$. Hence the AAS and TDS methods are not dual, therefore when considering their duals DAS and TAS we get a means to surround both extreme fixpoints of $\overline{\text{Int}}$:

\[ \sim \]
\[ AKS \]
\[ \ell_{fp} \]
\[ gfp \]
\[ DKS \]
Any of the AAS, TDS, DAS, TAS methods may yields a fixpoint $fp$ which is not the fixpoint $fp$ of interest. None of these methods can improve $fp$ to reach $fp$ or $gfp$, therefore a "fix-point improvement method" is necessary. It is our feeling that such a method could be designed only when considering that A-Gont possesses a richer structure (i.e. for particular applications).

Furthermore, in the AAS, TDS, DAS, TAS sequences the term of rank $n$ is computed only as a function of the term of rank $n-1$, hence these are "separate steps" methods. One can as well imagine to use "bound steps" methods, where the term of rank $n$ is computed as a function of the terms of rank $n-1$, $n-2$, ..., $n-k$. In this last case the Kleene's sequences may be used to compute the first $k$ terms. After $k$ steps more informations about the program would be available to heuristically accelerate the convergence so that the definition of A-int and D-int could be more refined.

Finally, going deeply into the comparism with numerical analysis methods, it is clear that some measure is necessary to control the accuracy of the result. Its definition would certainly also necessitate some additional properties of the abstract contexts.

10. Conclusion

It is our feeling that most program analysis techniques may be understood as abstract interpretations of programs. Let us point out global data flow analysis in optimizing compilers (Kildall[73], Morel and Renvoize[76], Schwartz[75], Ullman[75], Wegbreit[73], ..., type discovery (Cousot[76], Sintzoff[72], Tenenbaum[74], ...), program testing (Henderson [75], ...) symbolic evaluation of programs (Hewitt et al.[73], Karr[76], ...), program performance analysis (Wegbreit[76], ...), formalization of program semantics (Hoare and Lauer[74], Ligler[75], Manna and Shamir[75], ...), verification of program correctness (Floyd[67], Park[69], Sintzoff[75], ...), discovery of inductive invariants (Katz and Manna[76], ...), proofs of program termination (Sack[74], ...), program transformation (Sintzoff [76], ...), ...

There is a fundamental unity between all these apparently unrelated program analysis techniques: a new interpretation is given to the program text which allows to build an often implicit system of equations. The problem is either to verify that a solution provided by the user is correct, or to discover or approximate such a solution.

The mathematical model we studied in this paper is certainly the weakest which is necessary to unify these techniques, and therefore should be of very general scope. It can be considerably enriched for particular applications so that more powerful results may be obtained.

Acknowledgements

We wish to thank M. Sintzoff for stimulating discussions. We were very lucky to have F. Blanc do the typing for us.

11. References


Cousot[76]. Static determination of dynamic properties of generalized type unions. Submitted for publication. (Sept.)


Kam and Ullman[75]. Monotone data flow analysis frameworks. TR.169, C.S. Lab., Princeton Univ.

Karr[76]. Affine relationships among variables of a program. Acta Inf. 6, 133-151.


Ligler[75]. Surface properties of programming languages constructs. Int. Symp. on Proving and Improving Programs, (G. Huet and G. Kahn, eds.), IRISA, France.


Naur[65]. Checking of operand types in ALGOL compilers, BIT 5, 151-163.


Scott[71]. The lattice of flow diagrams. Symp. on Semantics of Programming Languages. Springer-Verlag Lecture Notes in Math. (E. Engeler, ed.), Vol. 188.


Sintzoff[73]. Vérifications d'assertions pour les fonctions utilisables comme valeurs affectant des variables extérieures. Int. Symp. on Proving and Improving Programs, (G. Huet and G. Kahn, eds.), IRIA, France.

Sintzoff[76]. Eliminating blind alleys from backtrack programs. Proc. of the third Int. Coll. on Automata, Languages and Programming, Edinburgh, (July).


Tennenbaum[74]. Type determination for very high level languages. NSO-3, Courant Inst. of Math. Sc., New York U., (Oct.).


13. Appendix

We note $\langle L, \leq, \leq, \sqsubseteq, \sqcup \rangle$ a complete $\leq$-semilattice $L$, with partial ordering $\leq$, supremum $\sqcup$ and infimum $\sqcap$. These definitions are given in Birkhoff[61].

Note: $L$ is a complete lattice.

(proof in Birkhoff[61], p. 49).

We take $f$ is isotone, $f$ is order-preserving of $f$ is monotone to be synonymous, and mean:

$\forall (x, y) \in L^2, (x \leq y) \Rightarrow \{f(x) \leq f(y)\}$

$\iff \forall (x, y) \in L^2, \{f(x \sqcup y) \geq f(x) \sqcup f(y)\}$

(H1): Let $F$ be an order-preserving function from the complete semi-lattice $\langle L, \leq, \sqsubseteq, \sqcup \rangle$ in itself.

(H2): Let $\bar{F}$ be an order-preserving function from the complete semi-lattice $\langle L, \leq, \sqsubseteq, \sqcup \rangle$ in itself.

(L1): The fixpoints of $F$ form a non-empty complete lattice with supremum $g$, infimum $\hat{x}$ such that:

$g = \bigcup \{x \mid (x \in L) \land (x \leq F(x))\}$

$\hat{x} = \bigcap \{x \mid (x \in L) \land (F(x) \leq x)\}$

(This result is proved in Tarski[55], pp. 286-287). Note that the fixpoints of $F$ need not form a sublattice of $L$.

We note $g$ and $\hat{x}$ the greatest and least fixpoints of $F$.

(H2): Let $\alpha$ and $\beta$ be such that:

$(H2.1) \alpha : L \rightarrow L$

$(H2.2) \beta : \bar{L} \rightarrow L$

$(H2.3) \alpha$ is order preserving

$(H2.4) \beta$ is order preserving

$(H2.5) \forall x \in L, x \leq \alpha(y(x))$

$(H2.6) \forall x \in L, x \leq \beta(x)($

(H3.1): (H1), (H2) and $\forall x \in L,$

$F(\alpha(x)) \leq \alpha(F(x))$

$\gamma(F(x)) \geq F(\gamma(x))$

(H3.2): (H1), (H2), and $\forall x \in L,$

$\gamma(F(x)) \geq F(\gamma(x))$

Proof:

$\forall x \in L,$

$F(\alpha(y(x))) \leq \alpha(F(y(x)))$ by $\gamma = \gamma(x)$ in H3.1

$F(x) \leq \alpha(F(x))$ from H2.5

$\gamma(F(x)) \geq \alpha(F(\gamma(x)))$ from H2.4

$\gamma(F(x)) \geq F(\gamma(x))$ H2.6 and transitivity.

$\forall x \in L,$

$\gamma(F(\alpha(x))) \geq F(\gamma(\alpha(x)))$ by $\alpha = \alpha(x)$ in H3.2

$\gamma(\alpha(x)) \leq \gamma(\alpha(x))$ H2.6

$F(\gamma(\alpha(x))) \geq F(x)$ P order preserving in (H1).

$\gamma(F(\alpha(x))) \geq F(x)$ transitivy

$\alpha(\gamma(F(\alpha(x)))) \geq \alpha(F(x))$ H2.3

$F(\alpha(x)) \geq \alpha(F(x))$ H2.5

Q.E.D.

Since H3.1 and H3.2 are proved by L2 to be equivalent, we choose:

(H3): (H3.1) or (H3.2)

(L3): Let $F : L \rightarrow L$ be an order-preserving function from the semi-lattice $\langle L, \leq, \leq, \sqcup \rangle$, in itself, $\leq$ and $\sqcup$ respectively the least and greatest fixpoints of $F$, then:

$\forall x \in L, \{g \sqcup F(x) \geq x\} \iff (g \geq x)$

(The dual of this result is proved in Park[69], pp. 66). By duality:

$\forall x \in L, \{\hat{x} \sqcap F(x) \leq x\} \iff (x \leq \hat{x})$
(T1) \( H_1, H_1, H_2, H_3 \) imply that the greatest fixpoints \( \gamma \) and \( \overline{\gamma} \) of \( F \) and \( \overline{F} \) are related by:
\[
(\alpha(\gamma) \preceq \overline{\gamma}) \quad \text{and} \quad (\overline{\gamma} \preceq \gamma(\gamma))
\]

Proof:

The existence of \( \overline{\gamma} \) is stated by (L1).

\[
\begin{align*}
\overline{\gamma} & \preceq \alpha(\gamma) \quad \text{trivially} \\
\overline{\gamma} & \preceq \alpha(\gamma) \quad \text{since} \quad \overline{\gamma} \preceq F(\gamma) \\
\overline{\gamma} & \preceq \alpha(\gamma) \quad \text{H3.1, \( \cup \) isotone, \( \preceq \) transitive} \\
\overline{\gamma} & \preceq \alpha(\gamma) \quad \text{H3.1} \\
\gamma(\overline{\gamma}) & \preceq \gamma(\alpha(\gamma)) \quad \text{H2.4} \\
\gamma(\overline{\gamma}) & \preceq \gamma(\alpha(\gamma)) \quad \text{H2.6, \( \preceq \) transitive.} \\
\end{align*}
\]

Q.E.D.

Replacing \( \langle \gamma, \overline{\gamma}, \preceq, \preceq, F, \overline{F}, \alpha, \gamma, H_3.1, H_2.4, H_2.6 \rangle \) respectively by \( \langle \dot{E}, \overline{E}, \preceq, \preceq, E, \overline{E}, F, \overline{F}, \dot{E} \rangle \), we get the "dual" theorem:

(\( T_2 \)) \( H_1, H_1, H_2, H_3 \) imply that the least fixpoints \( E \) and \( \overline{E} \) of \( F \) and \( \overline{F} \) are related by:

\[
(\gamma(\overline{E}) \preceq \overline{E}) \quad \text{and} \quad (\overline{E} \preceq \alpha(\overline{E}))
\]

According to Scott[7] a subset \( X \subseteq L \) is called directed if every finite subset of \( X \) has an upper bound (in the sense of \( \preceq \) belonging to \( X \). (An obvious example of a directed subset is a non-empty ascending chain.) A function \( f : D \rightarrow D \) is called continuous if whenever \( X \subseteq L \) is directed, then \( f(\cup\{x \mid x \in X\}) = \cup\{f(x) \mid x \in X\} \).

(H4): Let \( F \) be a continuous function from the complete semi-lattice \( \langle L, \cup, \preceq, \tau, \iota \rangle \) in itself.

(H4'): Let \( \overline{F} \) be a continuous function from the complete semi-lattice \( \langle L, \cup, \preceq, \tau, \iota \rangle \) in itself.

We note \( F^0(x) = x \) and \( F^{n+1}(x) = F(F^n(x)) \).

(L4): \( H_4(\overline{H}4) \) implies that \( F(\overline{F}) \) has a least fixpoint \( \overline{\iota} \) which is the limit \( \cup_{n=0}^{\infty} F^n(\iota) \) of the Kleene's sequence \( \iota \preceq F(\iota) \preceq \ldots \preceq F^n(\iota) \preceq \ldots \)

(The proof is easy to adapt from Kleene[52]'s proof of the first recursion theorem pp. 348-349).