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 denotation to describe computations in another universe of abstract objects, so that the results of abstract execution give some informations on the actual computations. An intuitive example (which we borrow from Sintzoff [72]) is the rule of signs. The text \(-1515 \times 17\) may be understood to denote computations on the abstract universe \(\{\ast\}, \{\odot\}, \{\div\}\) where the semantics of arithmetic operators is defined by the rule of signs. The abstract execution \(-1515 \times 17\) \(\Rightarrow -\odot \ast \odot \Rightarrow -\odot \ast \odot \Rightarrow \odot\), proves that \(-1515 \times 17\) 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. \(-1515 + 17 \Rightarrow -\odot + \odot \Rightarrow -\odot + \odot \Rightarrow \odot\)). 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.

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 :

\[
\text{n-succ, n-pred : Nodes} \rightarrow \text{2Nodes} \quad (m \in \text{n-succ}(n)) \Rightarrow (n \in \text{n-pred}(m))
\]

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

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

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

* Attache de Recherche au C.N.R.S., Laboratoire Associé n° 7.
** This work was supported by IRIA-SESORT under grants 72-035 and 76-160.
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 $\{\emptyset, T\}$ to it, and imposing the ordering $\frac{1}{2} \leq x \leq \frac{3}{2}$ for all $x \in S$.
- The semantic domain "Values" is a complete lattice which is the sum of the lattice Bool $= \{true, false\}^3$ and some other primitive domains.
- Environments are used to hold the bindings of identifiers to their values:

$$\text{Env} = \text{Ident} \rightarrow \text{Values}$$

We assume that the meaning of an expression $\text{expr}$ in the environment $e \in \text{Env}$ is given by $\text{val} \upharpoonright \text{Expr} (e)$ so that:

$$\text{val} : \text{Expr} \rightarrow \{\text{Env} \rightarrow \text{Values}\}$$

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

$$\text{val} \upharpoonright \text{Bexp} : \text{Bexp} \rightarrow \{\text{Env} \rightarrow \text{Booll}\}$$

- The state set "States" consists of the set of all information configurations that can occur during computations:

$$\text{States} = \text{Arcs} \times \text{Env}$$

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

$$\forall s \in \text{States}, s = (c(s), env(s)).$$

- We use a continuous conditional function $\text{cond}(b, e_1, e_2)$ equal to $1$, $e_1$, $e_2$ or $\top$ respectively as the value of $b$ is $1$, $true$, $false$ or $\top$. We also use $\text{if } b \text{ then } e_1 \text{ else } e_2 \text{ if }$ to denote $\text{cond}(b, e_1, e_2)$.

- If $e \in \text{Env}, v \in \text{Values}, x \in \text{Ident}$ then:

$$\text{val} (v/x, y) = \lambda y. \text{cond}(y = x, v, e(y)).$$

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

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

$$\text{n-state}(s) =$$

Let $n$ be $\text{end}(c(s))$, $e$ be $\text{env}(s)$ within case $n$ in

$\begin{align*}
\text{Assignments} & \quad \rightarrow \quad a-\text{succ} (n) \text{, if } \text{val} \upharpoonright \text{Expr} (n) \upharpoonright (e) / (n) > \\
\text{Tests} & \quad \rightarrow \quad \text{cond}(\text{val}(\text{test}(n) \upharpoonright e) / \text{Bexp}, \langle a-\text{succ}(n), e \rangle, \langle a-\text{succ}(f(n)), e \rangle) \\
\text{Junctions} & \quad \rightarrow \quad \text{val} \upharpoonright \text{Bexp} (n) \\
\text{Exits} & \quad \rightarrow \quad s
\end{align*}$

Each partial function $f$ on a set $S$ is extended to a continuous total function on the corresponding domain $S^\delta$ by $f(\emptyset) = 1$, $f(\top) = 1$ and $f(x) = 1$ if the partial function is undefined at $x$.

- Let $\text{Env}$ be the bottom function on $\text{Env}$ such that:

$$\forall s \in \text{Ident}^3, \text{Env}(s) = \{\text{Values}\}.$$

Let $I$-states be the subset of initial states:

$I$-states $= \{<a-\text{succ}(n), \text{Env} > | m \in \text{Entries} \}$
A "computation sequence" with initial state $i_0 \in I\text{-}states$ is the sequence:

$$s_n = n\text{-}state^n(i_0)$$

for $n = 0, 1, \ldots$

where $\text{identity}_n$ is the identity function and $\text{identity}_{n+1} = f \circ f_n$.

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:

$$n\text{-}state^\infty = \text{Y}_{\text{States} \rightarrow \text{States}} (\lambda F. (n\text{-}state \circ F))$$

where $\text{Y}_f$ denotes the least fixpoint of $f : D \rightarrow D$, Tarski [55].

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

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

where

$$F\text{-}cont : \text{Context-Vectors} \rightarrow \text{Context-Vectors}$$

is defined by:

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

$\text{Context-Vectors}$ is a complete lattice with union $\cup$ such that $Cv_1 \cup Cv_2 = \lambda r. (Cv_1(r) \cup Cv_2(r))$.

$F\text{-}cont$ is order preserving for the ordering $\preceq$ of $\text{Context-Vectors}$ which is defined by:

$$(Cv_1 \preceq Cv_2) \iff (\forall r \in \text{Arcs}, Cv_1(r) \preceq Cv_2(r))$$

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 $Cv$ is
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 \( A\text{-Cont} = \text{Arcs}^0 \rightarrow A\text{-Cont} \).

Whatever \((Cv', Cv'') \in A\text{-Cont}^2\) may be, we define:

\[
Cv' \oplus Cv'' = \lambda r. Cv'(r) \diamond Cv''(r)
\]

\[
Cv' \odot Cv'' = \{r \in \text{Arcs}^0, Cv'(r) \leq Cv''(r)\}
\]

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

\[<A\text{-Cont}, \leq, \sim, \doteq, \bot, \top> \] can be shown to be a complete lattice. The function:

\[
\text{Int} : \text{Arcs}^2 \rightarrow A\text{-Cont} = A\text{-Cont}
\]

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,
\[
(Cv' \odot Cv'') \rightarrow (\text{Int}(a, Cv') \leq \text{Int}(a, Cv''))
\]

The local interpretation of elementary program constructs which is defined by \( \text{Int} \) is used to associate a system of equations with the program. We define:

\[
\text{Int} : A\text{-Cont} \rightarrow A\text{-Cont} | \text{Int}(Cv) = \lambda r. \text{Int}(r, Cv)
\]

It is easy to show that \( \text{Int} \) is order-preserving. Hence it has fixpoints, \( \text{Tarski}' [55] \). Therefore the context vector resulting from the abstract interpretation \( I \) of program \( P \), which defines the global properties of \( P \), may be chosen to be one of the extreme solutions to the system of equations:

\( Cv = \text{Int}(Cv) \).

5.2 Typology of Abstract Interpretations

The restriction that "\( A\text{-Cont} \)" must be a complete semi-lattice is not drastic since Mac Neille[37] showed that any partially ordered set \( S \) can be embedded in a complete lattice so that inclusion is preserved, together with all greatest lower bounds and

\[
\text{Examples:}
Kildall[73] uses \((\rightarrow, \sim)\), Wegbreit[75] uses \((\rightarrow, \doteq)\). Tenenbaum[74] uses both \((\rightarrow, \doteq)\) and \((\doteq, \sim)\).

5.3 Examples
5.3.1 Static Semantics of Programs

The static semantics of programs we defined in section 4 is an abstract interpretation:

\[
I_{ss} = <\text{Contexts}, \leq, \text{Env}, \emptyset, n\text{-context}>
\]

where Contexts, \( \leq \), Env, \( \emptyset \), n-context, Context Vectors, \( \leq \), F-Context respectively correspond to \( A\text{-Cont}, \leq, \sim, \doteq, \bot, \top, \text{Int}, A\text{-Cont}, \leq, \sim, \text{Int} \).

5.3.2 Data Flow Analysis

Data flow analysis problems (see references in Ullman[75]) may be formalized as abstract interpretations of programs.

"Available expressions" give a classical example. An expression is available on arc \( r \), if whenever control reaches \( r \), the value of the expression has been previously computed, and since the last computation of the expression, no argument of the expression has had its value changed.
The determination of available expressions, back-
...dominators, intervals, ... requires a forward sys-
tem of equations. Some global flow problems, nota-
...the live variables and very busy expressions re-
quire propagating information backward through the
program graph, they are examples of backward sys-
tems of equations.

6.3.3 Remarks

Our formal definition of abstract interpretations
has the completeness property since the model en-
sure 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 defined
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 refe-
rence to the program syntax (e.g., the algorithm for
finding available expressions is justified by rea-
siong 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 \( \mathcal{I} = \langle \mathcal{A}, \mathcal{C}, \mathcal{V}, \mathcal{S}, \mathcal{T}, \mathcal{I}, \mathcal{Int} \rangle \) of a program is consistent with a "concrete"
interpretation \( \mathcal{I} = \langle \mathcal{C}, \mathcal{S}, \mathcal{T}, \mathcal{I}, \mathcal{Int} \rangle \) if the context vector \( \mathcal{CV} \) resulting from \( \mathcal{I} \) is a cor-
rect approximation of the particular context \( \mathcal{CV} \) result-
ing from the more refined interpretation \( \mathcal{I} \). This
may be rigorously defined by establishing a corres-
pondence \( (\alpha : \text{abstraction}) \) between concrete and ab-
stract context vectors, and inversely \( (\gamma : \text{concreti-
}}

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

\[
\begin{align*}
\forall (a, \tilde{x}) \in \text{Arcs} \times \text{A-Cont}, \\
\gamma(\text{Int}(a, \tilde{x})) & \geq \text{Int}(a, \gamma(\tilde{x})) \\
\end{align*}
\]

6.5 and

\[
\begin{align*}
\forall (a, x) \in \text{Arcs} \times \text{C-Cont}, \\
\text{Int}(a, \gamma(x)) & \geq \gamma(\text{Int}(a, x)) \\
\end{align*}
\]

These two hypothesis are in fact equivalent (lemma
1.2 in appendix 12). The following schema illus-

Suppose we want to compute the concrete output con-
text \( \mathcal{C}_0 \) (associated with arc \( a \)) resulting from con-
crete input contexts \( \mathcal{C}_1 : \mathcal{C}_0 = \text{Int}(a, \mathcal{C}_1) \). We can
as well approximate this computation in the abstract
universe, and get \( \mathcal{C}_0' = \gamma(\text{Int}(a, \gamma(\mathcal{C}_1))) \). 6.5 requires
\( \mathcal{C}_0' \) to contain at least \( \mathcal{C}_0 \), that is \( \mathcal{C}_0' \subseteq \mathcal{C}_0 \). On
the contrary we do not require \( \mathcal{C}_0' \) to contain at most
\( \mathcal{C}_0 \), that is \( \mathcal{C}_0 \subseteq \mathcal{C}_0' \) is not compulsory.

We will say that \( \mathcal{I} \) is a refinement of \( \mathcal{I} \), or that
\( \mathcal{I} \) is an abstraction of \( \mathcal{I} \), denoted \( \mathcal{I} \leq (\alpha, \gamma(\mathcal{I})) \), if
and only if there exist \( \alpha \) and \( \gamma \) satisfying hypothe-
sis 6.1 to 6.3.

Note that \( \mathcal{I} \leq (\alpha, \gamma(\mathcal{I})) \) imposes a local consistency
of the interpretations \( \mathcal{I} \) and \( \mathcal{I} \), at the level of pri-
mitive language constructs (6.5). Theorems T1 and
T2 of Appendix 12 then prove 6.0 which defines the
global consistency of \( \mathcal{I} \) and \( \mathcal{I} \) at the program level.

In particular if we take

\[
\mathcal{I}_{GS} = \langle \text{Contexts}, \mathcal{G}, \mathcal{S}, \mathcal{Int}, \mathcal{C} \rangle
\]

any abstract interpretation \( \mathcal{I} \) of \( \mathcal{P} \), consistent with
\( \mathcal{I}_{GS} \), \( \mathcal{I}_{GS} \leq (\alpha, \gamma(\mathcal{I})) \) is consistent with the seman-
tics of \( \mathcal{P} \), which implies :

\[
\forall (a, \gamma(\mathcal{I})) \text{ be the result of } \mathcal{I}, \\
\forall (a, \mathcal{S}) \in \text{States} \mid q < a > = n \cdot \text{scale}(i_a) \\
\implies (o \in \gamma(\mathcal{C}(q)))
\]

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

\[
\mathcal{I}_{DS} = \langle \text{Pred, or, } \Rightarrow, \text{ true, false, n-pred} \rangle
\]

242
where n-pred defines Floyd[67]'s strongest post condition:

\[ n\text{-pred}(r, P_v) = \]

\[ \text{let}\( s = \text{origin}(r) \), \( \langle p \rangle = \text{a-pred}(\text{origin}(r)) \) \text{within case } n \text{ in }\]

\[ \begin{align*}
\text{Entries} & \quad = \Rightarrow (\forall x \in \text{Ident}, x = s) \\text{true} & \\
\text{Junctions} & \quad = \Rightarrow \quad (P_v(q)) \\
\text{Tests} & \quad = \begin{cases} & \text{case } \mathcal{I} \text{ in } \mathcal{I} \text{ within } \\
& \quad \left( a\text{-succ}(n) \Rightarrow P_v(p) \quad \text{and} \quad \text{test}(n) \right) \\
& \quad \left( a\text{-succ}(n) \Rightarrow P_v(p) \quad \text{and} \quad \text{not} \text{test}(n) \right) & \\
\text{ esac} & \end{cases} \\
\text{Assignments} & \quad = \begin{cases} & \text{let } (P \text{ be } P_v(p)), (x \text{ be } \text{id}(n)), \\
& \quad \left( e \text{ be } \text{expr}(n) \right) \text{ within } \\
& \quad (\forall v \in \text{Values}) | \left[ P_v[v/x] \right] \text{ and } x = e[v/x] \text{ within } \end{cases} & \text{ esac} \\
\end{align*} \]

The "invariants" of the program are defined by the least fixpoint of n-pred (least for ordering \( \subseteq (\equiv) \), so that an invariant implies any other correct assertion).

The deductive semantics is easily validated by proving that \( I_{DG} \subseteq (\alpha, \gamma) I_{DG} \) where:

\[ \begin{align*}
\alpha & : \text{Contexts} \rightarrow \text{Pred} \\
& = \mathcal{C}. (\text{or } \text{ and } x = e(x)) \quad \forall \mathcal{C} \in \text{Ident} \\
\gamma & : \text{Pred} \rightarrow \text{Contexts} \\
& = \mathcal{P}. (e | P[e(x)/x], x \in \text{Ident}) \\
\end{align*} \]

The main point is to justify Hoare[67]'s proof rules by showing:

\( \{ \forall a \in \text{Arcs}, \forall p \in \text{Pred}, \alpha(\text{m-context}(a, \gamma(P_v))) \Rightarrow n\text{-pred}(a, P_v) \} \)

See Hoare and Lauer[74], Ligler[75]. In particular Ligler[75] shows clearly that the proof can be done only when considering realizable Contexts and programs involving "clean" basic constructs (e.g., constructs excluding non-termination, errors, side-effects, sharing between identifiers, ...).

Once \( I_{DG} \subseteq (\alpha, \beta) I_{DG} \) has been proved, we know that the deductive semantics gives a valid proof technique, which will never permit a false theorem to be deduced:

\( \forall q \in \text{Arcs}, \forall P_v \rightarrow \text{result of } I_{DG} \)

\( \{ \exists n \geq 0, \exists i_s \in I\text{-states} | \langle q, e \rangle = n\text{-state } I_{DG}(s) \Rightarrow (P_v(q) \Rightarrow \alpha(e)) \} \)

7. The Lattice of Abstract Interpretations

The relation \( \leq \) comparing the levels of abstraction of two interpretations is a quasi-ordering since it is:

reflexive: \((I \leq (1, 1)I) \) where \( x = \lambda a \cdot x \cdot x \) is the identity function,

transitive: \((I \leq (\alpha_1, \gamma_1)I') \) and \((I' \leq (\alpha_2, \gamma_2)I'')) \) imply \( I \leq (\alpha_1 \circ \alpha_2, \gamma_2 \circ \gamma_1)I'' \).

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:

\[ \{ I \equiv (\beta I') \} \iff \{ \beta \text{ is an isomorphism between the algebras } I \text{ and } I' \} \]

The proof gives some insight in the abstraction process:

\[ 1\cdot I \leq (\beta I') \Rightarrow \{ I \leq (\beta, \beta^{-1})I' \text{ and } I' \leq (\beta^{-1}, \beta)I \} \]

2. reciprocally,

If \( I \leq (\alpha, \gamma)I' \), let \( \equiv (\alpha, \gamma) \) be the equivalence relation defined on \( I \) (properly speaking, on the set of abstract contexts of \( I \)) by:

\[ \{ x \equiv (\alpha_1)(y) \iff \{ \alpha_1(x) = \alpha_1(y) \} \]

\( \forall x' \in I' \), each equivalence class \( \gamma_i = \{ x \in I | \alpha_i(x) = x' \} \) has a least upper bound which is \( \gamma_i(x') \). Hence the projection \( \alpha_i \mid \gamma_i(I') \) of \( \alpha_i \)

on \( \gamma_i(I') \) is a bijection from the set \( \gamma_i(I') \) of representatives of the equivalence classes on \( I \).

Let us show now that under the hypothesis \( I \leq (\alpha_1, \gamma_1)I' \) and \( I' \leq (\alpha_2, \gamma_2)I' \), \( \alpha_1 \) is bijective:

\( \alpha_1 \mid \gamma_1(I') \) and \( \alpha_2 \mid \gamma_2(I') \) are bijections, hence \( \forall x' \in I' \), \( \forall x \in \text{Expr}(I') \) such that \( x' = (\alpha_1 \mid \gamma_1(I'))(x)(x) \). Likewise, \( x' \in \gamma_2(I') \Rightarrow \forall x \in I' \Rightarrow x(x) = (\alpha_2 \mid \gamma_2(I'))(x)(x) \).

Therefore, \( \forall x' \in I' \Rightarrow \forall x \in I' \Rightarrow x(x) = (\alpha_1 \mid \gamma_1(I'))(x)(x) \).

Thus \( (\alpha_1 \mid \gamma_1(I')) \mid (\alpha_2 \mid \gamma_2(I')) \) is a bijection between \( \gamma_2(I) \) and \( I' \). Since \( (\alpha_2 \mid \gamma_2(I'))^{-1} \) is a bijection between \( I \) and \( \gamma_2(I) \), the composition

\( (\alpha_1 \mid \gamma_1(I')) \circ (\alpha_2 \mid \gamma_2(I')) \circ (\alpha_2 \mid \gamma_2(I'))^{-1} \)

is a bijection between \( I \) and \( I' \), hence \( \alpha_1 \) is a bijection between \( I \) and \( I' \) which is trivially an algebraic morphism. \( \alpha_1 \) is isotope, its inverse \( \alpha_1^{-1} = \gamma_1 \) is isotope and \( \alpha_1(\text{Int}(a, X)) \)

\[ = \text{Int}'(a, \alpha_1(X)) \text{ Q.E.D.} \]

Let I be the set of abstract interpretations of a program, if equivalent interpretations are identified, the quasi-ordering \( \leq \) becomes a partial ordering.

In particular, we can restrict I to be set of interpretations which abstract \( I_{DG} \). I is then a lattice, (with ordering \( \leq \)) which is isomorphic with a subset of the lattice of equivalence relations on Contexts.

Example:

Let P be a program with a single integer variable, (the generalized is obvious). Environments will be integers (the value of the variable). Contexts are sets of integers (the set of values at some program point).

A context S may be abstracted by a closed interval \( \alpha(S) = [\text{min}(S), \text{max}(S)] \). Where S is infinite the bounds will eventually be -\( \infty \) and \( \infty \).

\( \forall a \in S, b \in S \) \rightarrow \( \{ x | a \leq x \leq b \} \). The abstract contexts are then, (Cousot[76]):

243
8. Abstract Evaluation of Programs

The system of equations:

\[ C_v : \text{Int}(C_v) \]

resulting from an interpretation \( I = \langle \text{A-Cont}, \preceq, \prec, \tau, i, \text{Int} \rangle \) 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):

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

8.1 Correctness

If \( \text{Int} \) is supposed to be a complete morphism (i.e. infinitely distributive over \( \preceq \)) then \( C_v \) 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 \( C_v \) 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, \( C_v \) is an approximation \( \langle C \rangle \) of the least fixpoint (e.g. Knuth and Ulman[75]).

8.2 Termination

The abstract evaluation terminates iff Kleene's sequence is finite. This may be the case because A-Cont is finite (e.g. type checking in ALGOL 60, Mauro[65]), or a finite subset only is to be considered for any particular program (e.g. type checking in ALGOL 68), or A-Cont may be of finite length \( m \) (the length of any strictly increasing chain is bounded by \( m \), Kildall[73], Wegbreit[75]) or A-Cont may be used to generate 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 (Sinzloff [75]) or heuristics (Katz and Manul[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 $A$-Cont be the lattice $\mathbb{R}^+$ of positive real numbers augmented by the upper bound $\infty$, with natural ordering $\leq$. The abstract interpretation:

$$\lambda = <\mathbb{R}^+, \max, \leq, 0, \infty, \text{Kir}>$$

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

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

Let $n$ be origin(r) within case $n$ in Entries $\Rightarrow 1$ {unique entry node}

Junctions $\cup$ Assignments $\Rightarrow \Sigma$ \quad \text{Cv}(p)

Tests $\Rightarrow$

\begin{align*}
\text{case } r \text{ in } a &- \text{succ-}(n) \Rightarrow \text{Cv}(\text{pre}(n) \cdot) \\
(1 - \text{Probes}(\text{test}(n) = \text{true}) &- \text{Cv}(\text{pre}(n) \cdot)
\end{align*}

\text{esac}

\text{esac}

The main difficulty is to obtain the probability $\text{Probes}(\text{test}(n) = \text{true})$ of taking the true path at a test node $n$. Suppose the values of these probabilities can be determined (from hypothesis on the input data).

For fixed probabilities, the function $\text{Kir}$ is clearly continuous (although it is not a complete morphism) since

if $\text{Cv}_0 \leq \text{Cv}_1 \leq \ldots \leq \text{Cv}_n \leq \ldots$

then $\max_{i=0}^{\infty} (\Sigma \quad \text{Cv}_i(p)) = \Sigma \quad (\max_{i=0}^{\infty} (\text{Cv}_i(p)))$

and $\max_{x} (n_1 \cdot x) = (\max_{x} (n_1)) \cdot x$.

The least fixpoint of $\text{Kir}$ is the limit of Kleene's sequence (the length of the sequence is in general infinite):

- Let $P$ be the program "begin L: go to L end". The number $n$ of iterations in the loop is given by the minimal solution to the equation $n = n + 1$ which is the limit of $0 + 1 + 1 + 1 + \ldots$

- Let $P$ be the program "begin while T do I end". The number $n$ of times the expression $T$ is tested is given by the minimal solution to the equation $n = 1 + q + n$ where $q$ is the probability of $T$ to be true. $n$ may be determined by the limit of Kleene's sequence:

$$0 + 1 + q + q^2 + \ldots + q^k + \ldots$$

which is an infinite series. Its sum is $\frac{1}{1 - q}$.

This abstract interpretation leads to a system of linear equations. Kleene's sequence corresponds to the Jacobi's iterative method (for numerical coefficients).

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 $\overline{I}$ ($1 \leq \overline{I}$ may be used for that purpose (e.g., Tennenbaum[74a]). It is often better to make approximations in $\overline{I}$, 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 $I = \langle A$, Cont, $\leq, \gamma, \text{Int} \rangle$ be an interpretation of $P$. When the least fixpoint $\text{Cv}$ of $\text{Int}$ is unreachable, we look for an upper bound $\overline{\text{UB}}$ of $\text{Cv}$, since according to the correctness requirement

$0, \text{Cv} \leq \gamma(\text{Cv})$ and $\overline{\text{UB}} \leq \overline{\text{UB}}$.

9.1.1 Increasing Approximation Sequence

Let $\text{A-Int} : A$-Cont $\rightarrow A$-Cont be such that:

9.1.1.1 $(\forall \alpha \geq 0. C = \text{A-Int}^{(0)}(\alpha) \text{not} \text{Int}(C) \leq C)$

$\iff (C \leq \text{A-Int}(C) \leq \text{A-Int}(C))$.

9.1.1.2 Every infinite sequence $I, \text{A-Int}(C), \ldots, \text{A-Int}(\text{Cv})$, $\ldots$ is not strictly increasing.

The approximation sequence $S_0, \ldots, S_n, \ldots$ is recursively defined by:

$$S_0 = I$$

$$S_{n+1} = \begin{cases} \text{if not}(\text{Int}(S_n)) \Rightarrow S_n \\ \text{else} \quad S_n \end{cases}$$

We now prove that $\exists m$ finite such that:

$S_0 \leq S_1 \leq \ldots \leq S_m = S_{m+1} \leq \ldots$

Let $m$ be the least natural number (eventually infinite) such that $S_m = S_{m+1}$. $\forall k \in [0, m]$, we know from 9.1.1.3 that $\text{not} \text{Int}(S_k) \leq S_k$. Whence by definition of the ordering $\preceq, S_k \neq \text{Int}(S_k) \preceq S_k$.

Since $S_k \leq \text{Int}(S_k) \preceq S_k$, is always true, we can state that $S_k \preceq \text{Int}(S_k) \preceq S_k$. Besides not(Int(S_k) $\preceq S_k$) and 9.1.1.1 imply:

$$S_{k+1} = \text{A-Int}(S_k) \preceq \text{Int}(S_k) \preceq S_k$$

and therefore we conclude $S_{k+1} \succeq S_k$. $\forall k \in [1, m]$. Moreover 9.1.1.2 implies that $m$ is finite. Q.E.D.

Let $\text{Cv}$ be the least fixpoint of $\text{Int}$, it is the greatest lower bound of the set of $X \in A$-Cont such that $\text{Int}(X) \supset X$ (Tarski[55]) hence:

$$\forall X \in A$-Cont, $\text{Int}(X) \supset X$ $\Rightarrow (\text{Cv} \supset X)$$

Since $S_m = S_{m+1}$ we have $\text{Int}(S_m) \supset S_m$ and therefore $\text{Cv} \supset S_m$. $S_m$ is a correct approximation of $\text{Cv}$.
9.1.2 Generalization of Kleene's Ascending Sequence

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

As before, we define:

\[ 9.1.3.5 \quad \widehat{\lambda} \text{Int} = \lambda \overline{\text{Int}} \cdot (\lambda q \cdot A \text{-int}(q, \text{Cv})) \]

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

Let us consider a sequence $S = \overline{\text{S}}$.
Let us note \([a, b]\) where \(a \leq b\) the predicate \(a \leq x \leq b\). The system of equations corresponding to the example is:

1. \(C_0 = [1, 1]\)
2. \(C_1 = [1, 1]\)
3. \(C_2 = [2, 101]\)
4. \(C_3 = [1, 100]\)
5. \(C_4 = [1, 100]\)

**Note:**
- \(C_1 \cup C_4 = [1, 101] \leq C_2 = [1, +\infty]\)
- \(C_5 = [101, +\infty]\)

Assignment statements are treated using an interval arithmetic (e.g., \([1, j] + [k, l] = [i+k, j+l]\) 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., \(1, j < 0, 0 < 0, 1 < \ldots\)) for the program \(x := 0\); while true \(d x := x + 1\), 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 \(\vee\) of intervals by:

\[\forall [i, j] \vee [k, l] = \begin{cases} i & \text{if } k < i; \\ j & \text{if } k > j; \\ \text{else if } i \leq j & \text{if } i \leq j \end{cases}\]

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

2. \(C_2 = C_2 \wedge (C_1 \cup C_4)\)

The corresponding approximation sequence is:

- \(C_0 = [1, 1]\)
- \(C_1 = [1, 1]\)
- \(C_2 = [2, 101]\)
- \(C_3 = [1, 100]\)
- \(C_4 = [1, 100]\)

**8.3 Finite Iterative and Decreasing Approximation of the Least Fixpoint Starting from an Upper Bound**

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

\(\overline{\text{Int}}(S)_m \geq \overline{\text{Int}}(S)_m \geq \overline{\text{Int}}(S)_m \geq \overline{\text{Int}}(C)\)

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

**8.3.1 Decreasing Approximation Sequence**

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

\(\overline{\text{Int}}^{n+1}(S)_m \geq \overline{\text{Int}}^{n}(S)_m \geq \overline{\text{Int}}^{n}(S)_m \geq \overline{\text{Int}}(C)\)

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

\(\overline{\text{Int}}^{n+1}(S)_m \geq D \geq \overline{\text{Int}}(D)\)

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

**8.3.2 Truncated Decreasing Sequence**

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

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

Let \( D \text{-} \text{Int} : A^{\text{-} \text{Cont}} \to A^{\text{-} \text{Cont}} \) be such that:

9.3.2.1 \( \forall c \in A^{\text{-} \text{Cont}} \}
\[ (c \geq \text{Int}(c)) \implies (c \geq D \text{-} \text{Int}(c) \geq \text{Int}(c)) \]

9.3.2.2 \( \forall c \in A^{\text{-} \text{Cont}}, \) every infinite sequence $c_0, D \text{-} \text{Int}(c_0), \ldots$ is strictly decreasing.

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

9.3.2.3
\[
S'_0 = S_m
\]
\[
S'_{n+1} = \begin{cases} S'_n \text{ if } (S'_n \neq \text{Int}(S'_n)) & \text{and} \ (S'_n \neq D \text{-} \text{Int}(S'_n)) \text{ then } D \text{-} \text{Int}(S'_n) \\ S'_n & \text{else} \end{cases}
\]

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

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

9.3.2.4
\[
S'_0 \geq S'_m \geq \text{Int}(S'_0) \geq \text{Int}(S'_m) \geq \text{Int}(S'_p)
\]

Then applying 9.3.2.1 we have:

\[
S'_p \geq D \text{-} \text{Int}(S'_p) = S'_1 \geq \text{Int}(S'_0) \geq \text{Int}(S'_p)
\]

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

\[
S'_0 = S'_1 \geq \text{Int}(S'_0) \geq \text{Int}(S'_p)
\]

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

\[
S'_{k-1} \geq S'_{k} \geq \text{Int}(S'_{k-1}) \geq \text{Int}(S'_p)
\]

Since \( \text{Int} \) is order preserving we have:

The limit of the descending sequence \( S'_0 = \tilde{S}, \ldots, S'_p = D \text{-} \text{Int}(\tilde{S}) \), \( \ldots \) is an upper bound of the greatest fixpoint of \( \text{Int} \).

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 \( D \text{-} \text{Int} \) by local modifications to \( \text{Int} \):

9.3.4.1 \( \Delta : A^{\text{-} \text{Cont}} \times A^{\text{-} \text{Cont}} \to A^{\text{-} \text{Cont}} \)

9.3.4.2 \( \forall (C, C') \in A^{\text{-} \text{Cont}} \)
\[ (C \geq C') \implies (C \geq C \Delta C' \geq C') \]

9.3.4.3 Every infinite sequence \( s_0, s_1, \ldots \) 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 \( C_0, C_1, \ldots \) not strictly decreasing.

The approximation interpretation \( D \text{-} \text{Int} : A^{\text{Arcs}} \times A^{\text{-} \text{Cont}} \to A^{\text{-} \text{Cont}} \) is defined by:

9.3.4.4
\[
D \text{-} \text{Int} = \lambda q, C_0 \text{. if } q \in W^{\text{Arcs}} \text{ then } C_0(q) \text{ if } \text{Int}(q, C_0) \text{ else } \text{Int}(q, C_0)
\]

This definition of \( D \text{-} \text{Int} \) trivially satisfies the requirement 9.3.2.1 since \( \forall C_0 \in A^{\text{-} \text{Cont}} \) with property \( C_0 \geq \text{Int}(C_0) \) implies \( C_0(q) \geq \text{Int}(q, C_0) \).

\( W^{\text{Arcs}} \) is a context. If \( q \in W^{\text{Arcs}} \) then 9.3.4.2 implies that \( C_0(q) \geq C_0 \Delta \text{Int}(q, C_0) = C_0 \Delta \text{Int}(q, C_0) \geq \text{Int}(q, C_0) \).

Otherwise, if \( q \notin W^{\text{Arcs}} \) then \( C_0(q) \geq \text{Int}(q, C_0) \).

\( \text{Int}(q, C_0) = D \text{-} \text{Int}(q, C_0) \).

Hence \( C_0 \geq D \text{-} \text{Int}(C_0) \).

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

(2) \( C_2 = C_2 \triangle (C_1 \cup C_4) \)

The descending approximation sequence is:

- \( C_2 = C_2 \triangle (C_1 \cup C_4) \)
  - \( [1, +\infty] \triangle ([1, 1] \cup [2, 101]) \)
  - \( = [1, +\infty] \triangle [1, 101] \)
- \( \times C_2 = [1, 101] \)
- \( C_3 = C_2 \cap [-\infty, 100] \)
  - \( = [1, 101] \cap [-\infty, 100] = [1, 100] \)
  - stop on that path.
- \( C_5 = C_2 \cap [101, +\infty] \)
  - \( = [1, 101] \cap [101, +\infty] = [101, 101] \)
  - 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.

### 9.6 Dual Approximation Methods

The lattice \( A_{\text{cont}} \) may be partitioned as follows:

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

When \( X \triangle Y \) 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 \( \{X \mid X \geq \int\int(X)\} \), so that their limit \( S_m \) is greater than \( \llp \).

It is clear that the ascending approximation sequence AAS when starting from \( 1 \) leads to an upper bound of the least fixpoint \( \llp \) of \( \int\), and the truncated descending sequence TDS when starting from \( \top \) 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 \( \int\).

249
Any of the AAS, TDS, DAS, TAS methods may yield a fixedpoint fp which is not the fixedpoint lfp or gfp of interest. None of these methods can improve fp to reach lfp or gfp, therefore a "fixedpoint improvement method" is necessary. It is our feeling that such a method could be designed only when considering that A-cont 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

Acknowledgments

We wish to thank M. Sintoff 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.)


Holec[67]. An axiomatic basis for computer pro-


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

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


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


13. Appendix

We note <L, u, ≤, τ, ⌈> a complete u-semilattice L, with partial ordering ≤, supremum τ and infimum ⌈. 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, or f is monotone to be synonymous, and mean:

∀(x, y) ∈ L², (x ≤ y) ⇒ (f(x) ≤ f(y))
⇐⇒ (∀(x, y) ∈ L², (f(x) v y) ≥ f(x) v f(y))

(H1): Let F be an order-preserving function from the complete semi-lattice <L, u, ��, ⌈, ⌈> in itself.

(H1): Let F be an order-preserving function from the complete semi-lattice <L, u, ��, ⌈, ⌈> in itself.

(L1): The fixpoints of F form a non-empty complete lattice with supremum g, infimum h such that:

g = v(x) | (x ∈ L) ∧ (x ≤ F(x))

h = ∨(x) | (x ∈ L) ∧ (F(x) ≤ 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 h the greatest and least fixpoints of F.

(H2): Let α and β be such that:

(H2.1) α : L → L

(H2.2) β : F → L

(H2.3) α is order preserving

(H2.4) β is order preserving

(H2.5) ∀x ∈ L, x ≤ α(β(x))

(H2.6) ∀x ∈ L, x ≥ α(β(x))

(H3.1) : (H1), (H2), and {∀x ∈ L, F(α(x)) ≥ α(F(x))}

(H3.2) : (H1), (H2), and {∀x ∈ L, γ(β(x)) ≥ F(γ(x))}

(H2) and (H3) imply (H1), (H2) and (H3).

Proof:
∀x ∈ L,
F(α(β(x))) ≥ α(F(β(x))) by x = γ(x) in H3.1
F(γ(β(x))) ≥ α(F(γ(β(x))) from H2.5
γ(β(x)) ≥ F(α(β(x))) from H2.6 and transitivity.
F(x) ≥ F(γ(β(x))) in (H1).
γ(β(x)) ≥ F(α(β(x))) transitivity
α(F(α(β(x))) ≥ α(F(x)) H2.3
α(F(x)) ≥ α(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 → L be an order-preserving function from the semi-lattice <L, u, ��, ⌈, ⌈> in itself, and g and h respectively the least and greatest fixpoints of F, then:

∀x ∈ L, {g = F(x) ≥ x} ⇐⇒ (g ≥ x)

(The dual of this result is proved in Park[69], pp. 66). By duality:
∀x ∈ L, {h = F(x) ≤ x} ⇒ (h ≤ x)
(T1): $H_1, H_1, H_2, H_3$ imply that the greatest fix-points $g$ and $\bar{g}$ of $F$ and $\bar{F}$ are related by:

\[(\alpha(g) \leq \bar{g}) \text{ and } (g \leq \gamma(\bar{g}))\]

**Proof:**

The existence of $g$ and $\bar{g}$ is stated by (L1).

\[
\begin{align*}
\bar{g} & \leq \alpha(g) \quad \text{trivially} \\
\bar{g} & \leq \alpha(F(g)) \leq \alpha(g) \quad \text{since } \bar{g} = F(g) \\
\bar{g} & \leq F(\alpha(g)) \leq \alpha(g) \quad H3.1, \text{ isotone, } \leq \text{ transitive} \\
\bar{g} & \leq \alpha(g) \quad \text{L3}
\end{align*}
\]