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

Patrick Cousot* and Radhia Cousot**
Laboratoire d'Informatique, U.S.M.G., BP. 53
38041 Grenoble cedex, France

1. Introduction

A program denotes computations in some universe of objects. Abstract interpretation of programs consists in using that denotation to describe computations in another universe of abstract objects, so that the results of abstract execution give some informations on the actual computations. An intuitive example (which we borrow from Sintoff [72]) is the rule of signs. The text -1515*17 may be understood to denote computations on the abstract universe \((+, -), (\leq)\) where the semantics of arithmetic operators is defined by the rule of signs. The abstract execution \(-1515 \times 17 \Rightarrow -(+) \times (+) \Rightarrow (-) \times (+) \Rightarrow (-),\) 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 -(+) \times (+) \Rightarrow (-) \times (+) \Rightarrow (-),) 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 many fixpoints of that system, Tarski [55]. The abstractions 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, which are 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:

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

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", "Junctions" and "Exits" partition the set Nodes.

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

---

* Attaché 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.
3.2 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 \( S^0 \) the complete lattice obtained from \( S \) by adjoining \( \{\text{false}, \text{true}\} \) to it, and imposing the ordering \( \forall s \in S, \forall x \in S^0, x \leq s \) for all \( x \in S \).

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

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

- We assume that the meaning of an expression \( e \) in the environment \( e \in \text{Env} \) is given by \( \text{val} \models e \text{Expr} \) in domain \( \text{Expr} \) has the functionality:
  \[ \text{val} \models e \text{Expr} \models [\text{Env} \rightarrow \text{Values}] \]

- In particular the projection \( \text{val} \models \text{Expr} \) of the function \( \text{val} \models [\text{Env} \rightarrow \text{Values}] \) has the functionality:
  \[ \text{val} \models e \text{Expr} \models [\text{Env} \rightarrow \text{Boo}l] \]

- The state set "States" consists of the set of all information configurations that can occur during computations:
  \[ \text{States} = \text{Arcs} \times \text{Env} \]

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

- We will use the following functions:
  \[
  \text{origin, end} : \text{Arcs} \rightarrow \text{Nodes} \mid (\forall n \in \text{Nodes}, a = <\text{origin}(a), \text{end}(a))
  \]
  \[
  \text{a-succ}(\text{Nodes}) = \text{Arcs} \\
  \text{a-succ}(n) = \{n, m \mid m \in \text{n-succ}(n)\}
  \]
  \[
  \text{a-pred}(\text{Nodes}) = \text{Arcs} \\
  \text{a-pred}(n) = \{m, n \mid m \in \text{n-pred}(n)\}
  \]
  \[
  \text{a-succ-t} : \text{Tests} \rightarrow \text{Arcs} \\
  \text{a-succ-t}(n) = <n, \text{n-succ-t}(n)>
  \]
  \[
  \text{a-succ-f} : \text{Tests} \rightarrow \text{Arcs} \\
  \text{a-succ-f}(n) = <n, \text{n-succ-f}(n)>
  \]

  \text{Example}:

  \[
  \begin{array}{c}
  \text{X1=1} \\
  \end{array}
  \]

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

  \[ \text{n-state} : \text{States} \rightarrow \text{States} \]

  \[ \text{n-state}(s) = \]

  \text{let n be end(c(s)), e be env(s) within Assignments} \]

  \text{case n in Assignments} \]

  \[ <\text{a-succ}(n), \text{val} \models \text{expr}(n) \models (e)/\text{id}(n)>, \text{Tests} \]

  \text{cond(\text{val} \models \text{test}(n) \models e), \text{Expr}, <\text{a-succ-t}(n), e>, <\text{a-succ-f}(n), e>>} \]

  \text{Junctions} \]

  \text{Exits} \]

  \text{exact} \]

  (Each partial function \( f \) on a set \( S \) is extended to a continuous total function on the corresponding domain \( S^2 \) by \( f(i) = 1, f(x) = f(f(x) = 1 \).
A "computation sequence" with initial state $i_s \in I$-states is the sequence:
$$s_n = n\text{-state}^n(i_s)$$
for $n = 0, 1, \ldots$
where $\text{id}$ is the identity function and $\text{id}^* = f \iff f_n$.

- The initial to final state transition function:
$$n\text{-state}^\omega : \text{States} \to \text{States}$$
is the minimal fixpoint of the functional:
$$\lambda F. \ (n\text{-state} \circ F)$$
Therefore:
$$n\text{-state}^\omega = Y_{\text{States} \to \text{States}}(\lambda F. \ (n\text{-state} \circ F))$$
where $Y(F)$ denotes the least fixpoint of $F$.

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 that to prove 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 $q \in \text{Arcs}$ of a program $P$ to be the set of all environments which may be associated to $q$ in all the possible computation sequences of $P$:
$$C_q \subseteq \text{Contexts} = 2^{\text{Env}}$$
$$C_q = \{ e | (\exists n \geq 0, \exists j_s \in I\text{-states} | <q, e> = n\text{-state}^n(j_s) \}$$

The context vector $C_v$ associates a context to each of the program points of a program $P$:
$$C_v \subseteq \text{Context-Vectors} = \text{Arcs}^0 \to \text{Contexts}$$
$$C_v = \lambda q. \ (e | (\exists n \geq 0, \exists j_s \in I\text{-states} | <q, e> = n\text{-state}^n(j_s))$$

According to the semantics of programs, the context $C_v(r)$ associated to arc $r$ is related to the contexts $C_v(q)$ at arcs $q$ adjacent to $r$, $(end(q) = \text{origin}(r), q = \text{pred}(r))$. From the definition of the static transition function we can prove the equation:
$$C_v(r) = n\text{-context}(r, C_v)$$
where
$$n\text{-context} : \text{Arcs}^0 \times \text{Context-Vectors} \to \text{Contexts}$$
is defined by:
$$n\text{-context}(r, C_v) = \cases{ origin(r) \in \text{Entries} \subseteq (\text{Env}) \\
\bigcup_{q \in \text{clos}(r)} e \in \text{env-on}(r)(n\text{-state}(<q, e>)) \cup \text{Assigns} \cup \text{Tests} \cup \text{Junctions} }$$
$$\text{clos}(r) = \{ q \in \text{pred}(\text{origin}(r)) \}$$
(We define $\text{env-on} : \text{Arcs}^0 \to (\text{States} \to (\text{Env}))$ to be
$$\lambda r. (\lambda s. \text{cond}(r = \text{ex}(s), (\text{env}(s)), \text{id}))$$. Since the equation $C_v(r) = n\text{-context}(r, C_v)$ must be valid for each arc $r$, $C_v$ is a solution to the system of "forward" equations:
$$C_v = n\text{-context}(C_v)$$
where
$$n\text{-context} : \text{Context-Vectors} \to \text{Context-Vectors}$$
is defined by:
$$n\text{-context}(C_v) = \lambda r. n\text{-context}(r, C_v)$$

Context-Vectors is a complete lattice with union $\cup$ such that $C_v \subseteq C_v \subseteq C_v \subseteq C_v$.

$C_v$ is order preserving for the ordering $\leq$ of Context-Vectors which is defined by:
$$C_v \subseteq C_v \iff (\forall e \in \text{Arcs}, C_v(r) \subseteq C_v(r))$$

Hence it is known that $C_v$ 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 $\geq$ to the system of equations $X = n\text{-context}(X)$, $C_v \subseteq X$. Tarski[55] shows that this property uniquely determines $C_v$ as the least fixpoint of $n\text{-context}$. Thus $C_v$ can be equivalently defined by:
$$D_1 : C_v = \lambda q. \ (e | (\exists n \geq 0, \exists j_s \in I\text{-states} | <q, e> = n\text{-state}^n(j_s))$$
or
$$D_2 : C_v = Y_{\text{Context-Vectors}}(n\text{-context})$$

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

(a) $C_v(q)$ contains at least the environments $e$ which may be associated to $q$ during any execution of $P$:
$$\{ e \in C_v(q) | (\exists i_s \in I\text{-states} | <q, e> = n\text{-state}^i(j_s))$$

(b) $C_v(q)$ contains only the environments $e$ which may be associated to $q$ during an execution of $P$:
$$\{ e \in C_v(q) | (\exists i_s \in I\text{-states} | <q, e> = n\text{-state}^i(j_s))$$

$C_v$ is merely a static summary of the possible executions of the program. However, our definitions $\text{D}_1$ or $\text{D}_2$ 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
$$I = \langle A\text{-Cont}, \leq, s, i, t, \text{Int} \rangle$$
where the set of abstract contexts is a complete o-semilattice with ordering $\leq$, $(x \leq y) \iff (x \circ y = y)$. This implies that $A\text{-Cont}$ has a supremum $\top$. We suppose also $A\text{-Cont}$ to have an infimum $i$.  

240
This implies that A-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-Cont = Arca × A-Cont.

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

\[
Cv' \underset{\sim}{\subset} Cv'' = \lambda r. Cv'(r) \leq Cv''(r)
\]

\[
\sim = \lambda r. \top \text{ and } \lambda = \lambda r. \bot
\]

\(<A-Cont, \sim, \leq, \top, \bot>\) can be shown to be a complete lattice. The function:

\[
\text{Int} : A-Cont \rightarrow A-Cont
\]

defines the interpretation of basic instructions. If \(\{C(q) \mid q \in a-\text{prod}(n)\}\) 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 \(\text{Int}(r, C)\).

Int is supposed to be order-preserving:

\[
\forall a \in A-Cont, \forall (Cv', Cv'') \in A-Cont^2, \quad (Cv' \underset{\sim}{\subset} Cv'') \implies (\text{Int}(a, Cv') \leq \text{Int}(a, Cv''))
\]

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:

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

It is easy to show that \(\text{Int}\) is order-preserving. Hence it has fixpoints, [Tarski55]. 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 of the system of equations.

**Examples**:

Kildall[73] uses \((n, v, t)\), Wegbreit[75] uses \((v, t, t)\). Tenenbaum[74] uses both \((v, t, t)\) and \((n, v, t)\).

**5.3 Examples**

**5.3.1 Static Semantics of Programs**

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

\[
I_{gs} = \langle \text{Contexts, } v, \leq, \text{Env, } \emptyset, \text{ n-context} \rangle
\]

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

**5.3.2 Data Flow Analysis**
The determination of available expressions, back-
dominators, intervals, ... requires a forward sys-
tem of equations. Some global flow problems, nota-
bly the live variables and very busy expressions
require propagating information backward through
the program graph, they are examples of backward
systems of equations.

6.3.5 Remarks

Our formal definition of abstract interpretations
has the completeness property since the model en-
sures 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 refe-
rence to the program syntax (e.g. the algorithm for
finding available expressions is justified by reas-
oning 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 $I = \{A, Cont, \bar{a}, \bar{x},$
$I, \bar{c}, \bar{t}, \bar{C}, \bar{I}\}$ of a program is consistent with a "con-
tcrete" interpretation $I = \{C, Cont, \bar{a}, \bar{x},
I, \bar{c}, \bar{t}, \bar{C}, \bar{I}\}$ if the context vector $\bar{C}_I$ result-
ing from $\bar{I}$ is a cor-
respondence of the abstract context vector $\bar{C}_I$
resulting from the more refined interpretation $I$. This
may be rigorously defined by establishing a corre-
spondence ($\bar{a} : abstraction$) between concrete and ab-
stract context vectors, and inversely ($\bar{y} : concreti-
zation$), and requiring:

6.0 $\{\bar{C}_I \leq \bar{y}(\bar{C}_I)\}$ and $\{\bar{y}(\bar{C}_I) \leq \bar{C}_I\}$

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

If $f : D \rightarrow D'$ we note $\bar{D} = Arcs_D \rightarrow D$ and $\bar{D}' = Arcs_D \rightarrow D'$
and $\bar{P} : D \rightarrow D' = \lambda \bar{a}. f(d)(\bar{a})$.
We will suppose $\bar{a}$ and $\bar{y}$ to satisfy the following hypothesis:

6.1 $\bar{C} : C = A - Cont \rightarrow A - Cont$, $y : A - Cont \rightarrow C - Cont$
6.2 $\bar{a}$ and $\bar{y}$ are order-preserving
6.3 $\forall \bar{C} \in A - Cont, \bar{C} = \bar{y}(\bar{y}(\bar{C}))$
6.4 $\forall \bar{C} \in C - Cont, \bar{C} = \bar{y}(\bar{a}(\bar{C}))$

Intuitively, hypothesis 6.2 is necessary because context inclusion (that is property comparison)
must be preserved by the abstraction or concreti-
ization process. 6.3 requires that concretization
introduces no loss of information. It implies that $\bar{a}$ is surjection and $\bar{y}$ is injection.

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

\[
\begin{align*}
(\forall a, \bar{x} \in Arcs \times A - Cont, \\
\bar{y}(\bar{a}, \bar{x}) \in \bar{y}(\bar{y}(\bar{a}, \bar{x}))
\end{align*}
\]

These two hypothesis are in fact equivalent (lemma
2.2 in appendix 1). The following schema illus-
trates 6.5, i.e. the idea of abstract simulation of
concrete computations:

\[
\begin{array}{c}
\bar{C}_I \\
\bar{y}(\bar{a}) \\
\bar{C}_I \\
\bar{y}(\bar{a}) \\
\bar{C}_I \\
\bar{y}(\bar{a}) \\
\bar{C}_I \\
\bar{y}(\bar{a}) \\
\end{array}
\]

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

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

Note that $I \leq (a, \bar{y})I$ imposes a local consistency of the interpre-
tations $I$ and $I$, at the level of primiti-

In particular if we take

\[
I_{SS} = \langle Contexts, \bar{u}, \bar{C}, Env, \alpha, \alpha - context \rangle
\]

any abstract interpretation $\bar{I}$ of $P$, consistent with
$I_{SS} (I_{SS} \leq (a, \bar{y})I)$ is consistent with the seman-
tics of $P$, which implies:

\[
\forall \bar{C} \in Arcs, let \bar{C}_I be the result of $I$,
\forall n \in 0, \forall \bar{C}_I \in I - states | \langle q, e \rangle = n - state^n_i (i, \bar{C}_I) \\
\implies (\langle q, \bar{C}(q) \rangle)
\]

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

Example: Deductive Semantics of Programs
where $\text{npred}$ defines Floyd's strongest post condition:

$\text{npred}(r, P_v) =$

\[
\begin{align*}
\text{let } (a \text{ be origin}(r)) & \text{ or } (p \text{ be a-pred(origin}(r)))\text{ within} \\
\text{case } n \text{ in} \\
\text{Entries} & \Rightarrow (\forall x \in \text{Ident}, x = ^i\text{Values}) \\
\text{Junctions} & \Rightarrow (P_v(q)) \\
\text{Tests} & \Rightarrow \text{case } n \text{ in} \\
& \quad (a = \text{succ-ct}(n)) \Rightarrow P_v(p) \text{ and } \text{test}(n) \\
& \quad (a = \text{succ-ft}(n)) \Rightarrow P_v(p) \text{ and } \text{not test}(n) \\
\text{esac} \\
\text{Assignments} & \Rightarrow \\
\text{let } (P \text{ be } P_v(p)), (x \text{ be id}(n)), & \\
& (e \text{ be expr}(n)) \text{ within} \\
& (\exists v \in \text{Values} \mid P[v/x] \text{ and } x = e[v/x]) \\
\text{esac}
\end{align*}
\]

The "invariants" of the program are defined by the least fixpoint of $\text{npred}$ (least for ordering $\Rightarrow$), so that an invariant implies any other correct assertion.

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

$\alpha = \gamma = \text{X}$

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

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

is an equivalence relation. We have:

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

The proof gives some insight in the abstraction process:

$1 - [I \equiv (\beta)I'] \iff \{ (I \subseteq (\beta, \beta^{-1})I') \text{ and} \\
(I' \subseteq (\beta^{-1}, \beta)I) \}$

2 - reciprocally, if $I \subseteq (\alpha, \gamma)I'$, let $[\alpha]_I$ be the equivalence relation defined on $I$ (properly speaking, on the set of abstract contexts of $I$) by:

$\{ x \equiv (\alpha_1)(y) \iff \{ \alpha_1(x) = \alpha_1(y) \}$

$\forall x', y' \in I'$, each equivalence class $C_{x'} = \{ x \in I \mid \alpha_1(x) = x' \}$ has a least upper bound which is $\gamma_1(x')$. Hence the projection $\alpha_1 \mid \gamma_1(I')$ of $\alpha_1$ on $\gamma_1(I')$ is a bijection from the set $\gamma_1(I')$ of representatives of the equivalence classes on $I$.

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

$\alpha_1 \mid \gamma_1(I')$ and $\alpha_2 \mid \gamma_2(I)$ are bijections, hence $\forall x', y' \in I'$, $(\alpha_1 \mid \gamma_1)(x') = (\alpha_2 \mid \gamma_2)(y')$ such that
8. Abstract Evaluation of Programs

The system of equations:

\[
\text{Cy} : \int_{\text{Int}}(\text{Cy})
\]

resulting from an interpretation \( I = \langle \text{Int}, \cdot, \leq, \tau, i, \text{Int} \rangle \) 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):

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

8.1 Correctness

If \( \text{Int} \) is supposed to be a complete morphism (i.e. infinitely distributive over \( \cdot \)) then \( \text{Cy} \) is the least fixpoint of \( \int_{\text{Int}} \) (e.g. Kildall[75]), since in a semi-lattice of finite length, any distributive function is a complete morphism. Verify the reader...
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 \(\omega\), with natural ordering \(\leq\). The abstract interpretation:

$$I_p = (\mathbb{R}^+, \max, \leq, 0, \omega, \text{Kir})$$

may be used to derive the mean values of the counter.

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 $\overline{I}$ ($1 \leq I$) may be used for that purpose (e.g. Tennenbaum[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 Incremental Approximation
9.1.2 Generalization of Kleene's Ascending Sequence

When A-Cont satisfies the ascending chain condition one can choose A-Int to be 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 A-Int in 9.1.1 is global. We now indicate a way to construct A-Int by local modifications to Int.

Let \((q, r) \in \text{Arcs}^2\), we say that the context associated to \(q\) is dependent on the context associated to \(r\), if and only if:
\[
\{ C_{r} \in \text{Arcs} \mid \text{Int}(q, C_{r} \cap C) \neq \text{Int}(q, C_{r} \setminus C) \}
\]
This is because in a forward system of equations the context associated to \(q\) may only depend on the contexts associated with the immediate predecessor arcs of \(q\). In the system of equations \(C_{r} = \text{Int}(C_{r})\) we define a cycle to be a sequence \(q_1, \ldots, q_n\) of arcs such that \(T \in \{1, \ldots, n\}, C_{r}(q_{T+1}) \subseteq C_{r}(q_{T}) \cap \text{Arcs}\), and \(C_{r}(q_{n}) \subseteq C_{r}(q_{1}) \cap \text{Arcs}\). (e.g. in a forward interpretation a cycle corresponds to a loop in the program.)

In any infinite strictly increasing Kleene's sequence \(C_{r}, \ldots, C_{r}\), since Arcs is finite there is some arc \(q\) for which the sequence \(C_{r}(q), \ldots, C_{r}(q)\), ... never stabilizes. Therefore \(q\) must belong to a cycle or the contexts associated to \(q\) transitively depend on the contexts associated to some other arc \(r\) which itself belongs to a cycle. The sequence of contexts associated to any arc of that cycle never stabilizes. In order to avoid this phenomenon, we introduce:

- The binary operation \(\vee\) called widening defined by:

9.1.3.1 \(\forall : \text{A-Cont} \times \text{A-Cont} \rightarrow \text{A-Cont}\)

9.1.3.2 \(\forall \subseteq \text{A-Cont} \times \text{A-Cont}\)

9.1.3.3 Every infinite sequence \(S_{n}, \ldots, S_{1}\) of the form \(S_{n} = C_{r}, \ldots, S_{i} = C_{r} \subseteq C_{r} \subseteq C_{r}\), ... (where \(C_{r} = \ldots, C_{r}\) are arbitrary contexts) is not strictly increasing.

- The set \(\text{W-arcs}\) of widening arcs, which is one of the minimal sets of arcs such that any cycle \(q_{1}, \ldots, q_{n}\) of the system of equations \(C_{r} = \text{Int}(C_{r})\) contains at least a widening arc: \(\forall T \in \{1, \ldots, n\}, q_{T} \in \text{W-arcs}\). (e.g. in a forward interpretation on a reducible program graph, \(\text{W-arcs}\) may be chosen to be the set of exit arcs of the junction nodes which are interval headers. On irreducible graphs an arbitrary choice has to be made so that any loop of the program goes through a widening arc.)

- The approximate interpretation \(A-\text{Int} : \text{Arcs} \times \text{A-Cont} \rightarrow \text{A-Cont}\) defined by:

9.1.3.4 \(A-\text{Int} = \lambda (q, C_{r})\) if \(q \in \text{W-arcs}\) then \(\text{Int}(q, C_{r}) \vee \text{Int}(q, C_{r})\) else \(\text{Int}(q, C_{r})\)

As before, we define:

9.1.3.5 \(A-\text{Int} = \lambda (q, C_{r})\) if \(q \in \text{W-arcs}\) then \(\text{Int}(q, C_{r}) \vee \text{Int}(q, C_{r})\) else \(\text{Int}(q, C_{r})\)

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

Let us consider a sequence \(S_{n} = \ldots, S_{1}\) of widening arcs. We show that this sequence is increasing that is to say:

9.1.3.6 \(S_{n} \supseteq A-\text{Int}(S_{n})\), \(\forall n \geq 0\).

Trivially for \(n = 0\), \(S_{0} = \{q\} = A-\text{Int}(S_{0})\). For the induction step, suppose the result to be true for \(n \leq m\). Let us prove that:

If \(q \in \text{W-arcs}\), then \(A-\text{Int}(S_{m+1}) \supseteq A-\text{Int}(S_{m+1})\)

Finally \(S_{n+1}\) is an infinite sequence of widening arcs, \(\forall q \in \text{Arcs}\).

An infinite sequence \(S_{n} = \ldots, S_{1}\) cannot be strictly increasing since otherwise there would exist some widening arc \(q\) for which the sequence \(S_{n}(q), \ldots, S_{1}(q)\) would never stabilize thus contradicting 9.1.3.3.

We now prove 9.1.1.1 that is to say that:

9.1.1.1 \(\forall n \geq 0\), \(S_{n} = A-\text{Int}(S_{n})\)

implies

9.1.1.2 \(S_{n} \supseteq A-\text{Int}(S_{n})\), \(\forall q \in \text{Arcs}\)

\(\Rightarrow \) \(S_{n}(q) \subset A-\text{Int}(q, S_{n})\) (see 9.1.3.5)

If \(q \in \text{W-arcs}\), we have \(A-\text{Int}(q, S_{n}) = S_{n}(q) \supseteq A-\text{Int}(q, S_{n}) = \text{Int}(q, S_{n})\) by 9.1.3.2. If now \(q \notin \text{W-arcs}\) we must show:

9.1.3.3 \(S_{n}(q) \subset A-\text{Int}(q, S_{n})\) by 9.1.3.4 which is true, from 9.1.3.6, \(\forall q \in \text{Arcs}\).

9.1.3.5 \(A-\text{Int} = \lambda (q, C_{r})\) if \(q \in \text{W-arcs}\) then \(\text{Int}(q, C_{r}) \vee \text{Int}(q, C_{r})\) else \(\text{Int}(q, C_{r})\)

9.1.3.6 \(S_{n} \supseteq A-\text{Int}(S_{n})\), \(\forall n \geq 0\).

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

Let us consider a sequence \(S_{n} = \ldots, S_{1}\) of widening arcs. We show that this sequence is increasing that is to say:

9.1.3.6 \(S_{n} \supseteq A-\text{Int}(S_{n})\), \(\forall n \geq 0\).

Trivially for \(n = 0\), \(S_{0} = \{q\} = A-\text{Int}(S_{0})\). For the induction step, suppose the result to be true for \(n \leq m\). Let us prove that:

\(S_{n+1} \supseteq A-\text{Int}(S_{n})\)

Finally \(S_{n+1}\) is an infinite sequence of widening arcs, \(\forall q \in \text{Arcs}\).

An infinite sequence \(S_{n} = \ldots, S_{1}\) cannot be strictly increasing since otherwise there would exist some widening arc \(q\) for which the sequence \(S_{n}(q), \ldots, S_{1}(q)\) would never stabilize thus contradicting 9.1.3.3.

We now prove 9.1.1.1 that is to say that:

9.1.1.1 \(\forall n \geq 0\), \(S_{n} = A-\text{Int}(S_{n})\)

\(\Rightarrow \) \(S_{n}(q) \subset A-\text{Int}(q, S_{n})\) (see 9.1.3.5)

If \(q \in \text{W-arcs}\), we have \(A-\text{Int}(q, S_{n}) = S_{n}(q) \supseteq A-\text{Int}(q, S_{n}) = \text{Int}(q, S_{n})\) by 9.1.3.2. If now \(q \notin \text{W-arcs}\) we must show:

9.1.3.3 \(S_{n}(q) \subset A-\text{Int}(q, S_{n})\) by 9.1.3.4 which is true, from 9.1.3.6, \(\forall q \in \text{Arcs}\).

9.2 Examples: Bounds of Integer Variables

In a PASCAL program operating on arrays, the compiler should ensure that arrays are subscripted only by indices within bounds. For that purpose one can use the lattice \(\mathbb{L}\) of section 7. Let us take an obvious example:
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:

- (0) \(C0 = \{1\}\)
- (1) \(C1 = \{1\}\)
- (2) \(C2 = C1 \cup C4\)
- (3) \(C3 = C2 \cap [\infty, 100]\)
- (4) \(C4 = C3 \cap [1, 1]\)
- (5) \(C5 = C2 \cap [101, \infty]\)

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

- \([i, j] \vee [k, l] = \{i < k \text{ i.e.}\ {-}\infty \text{ else } i, i, j, k > j \text{ else } +\infty \text{ else } l\}\)

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

- (2) \(C2 = C2 \vee (C1 \cup C4)\)

The corresponding approximation sequence is:

- \(C1 = \{1\}\) for \(i \in [0, 3]\)
- \(C2 = C2 \vee (C1 \cup C4)\)
- \([1, 1] \vee ([1, 1] \cup [1, 1])\)
- \([1, 1] \vee [1, 1]\)
- \([1, 1] \cap [\infty, 100]\)
- \([1, 1] \cap [\infty, 100]\)
- \([1, 1] \cap [\infty, 100]\)
- \([1, 1] \cap [\infty, 100]\)
- \([2, 2]\)
- \([2, 2]\)
- \([1, 1] \vee ([1, 1] \cup [2, 2])\)
- \([1, 1] \vee [1, 2]\)
- \([1, 1] \vee [1, 2]\)
- \([1, \infty] \vee [\infty, \infty]\)
- \([1, \infty] \vee [\infty, 100]\)
- \([1, \infty] \vee [\infty, 100]\)

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.

### 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 \(S_\infty = A^{\downarrow}(\bar{S}_\infty)\) of the least fixpoint \(\bar{C}_V\) of of \(\bar{A}\) and \(\bar{X}\). Moreover \(\bar{A}^{\downarrow}(S_m) \subseteq S_m\). Since \(\bar{A}^{\downarrow}\) is order preserving, this implies that:

\[
S_m \supseteq \bar{A}^{\downarrow}(S_m) \supseteq \ldots \supseteq \bar{A}^{\downarrow}(S_m) \supseteq \ldots \supseteq \bar{C}_V.
\]

If \(S_m\) is not a fixpoint of \(\bar{A}\) and the above descending sequence is finite (e.g., the lattice \(\bar{A}^{\downarrow}\) satisfies the descending chain condition) its limit is a better approximation of \(\bar{C}_V\) 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:

\[
\bar{A}^{\downarrow}(S_m) \supseteq \bar{A}^{\downarrow}(S_m) \supseteq \ldots \supseteq \bar{A}^{\downarrow}(S_m) \supseteq \ldots \supseteq \bar{C}_V.
\]

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

\[
\bar{A}^{\downarrow}(S_m) \supseteq D \supseteq \bar{A}^{\downarrow}(D).
\]

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

#### 8.3.2 Trimmed Decreasing Sequence

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

\[
\exists n \geq 0 | \bar{A}(S_m) \supseteq D \supseteq \bar{A}^{\downarrow}(S_m)
\]

247
(However, we will not artificially truncate the decreasing sequence by imposing an arbitrary upper bound on n).

Let \( D \)-Int : A-Cont \rightarrow A-Cont \) be such that:

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

9.3.2.2 \( \forall C \in A-\text{Cont} \), every infinite sequence \( C, D-\text{Int}(C), \ldots, D-\text{Int}\)^n(C), \ldots \) is not strictly decreasing.

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

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

Let us now prove that the truncated decreasing se-

The limit of the descending sequence \( S'_0 = \tilde{t}, \ldots, S'_n = D-\text{Int}(\tilde{t}), \ldots \) 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 \( D-\text{Int} \) by local modifications to \( \text{Int} \):

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

9.3.4.2 \( \forall (C, C') \in A-\text{Cont}^2, \)
\[ (C \geq C') \implies (C \geq C \Delta C' \geq C') \]

9.3.4.3 Every infinite sequence \( s_0, \ldots, s_n, \ldots \) 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 \( D-\text{Int} : Arcs^3 \times A-\text{Cont} \rightarrow A-\text{Cont} \) is defined by:
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:

\[ C_2 = C_2 \triangle (C_1 \cup C_4) \]
\[ = [1, + \infty] \triangle ([1, 1] \cup [2, 101]) \]
\[ = [1, + \infty] \triangle [1, 101] \]
\[ * \]
\[ C_2 = [1, 101] \]

\( C_3 = [1, 101] \cap [-\infty, 100] \)
\[ * \]
\[ C_3 = [1, 101] \cap [-\infty, 100] = [1, 100] \]

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 \( \widetilde{\text{Cont}} \) may be partitioned as follows:

When \( X \geq Y \) 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 \( \{X \mid X \geq \widetilde{\text{Int}}(X)\} \), so that their limit \( S^p \) is greater than \( \mathbf{lfp} \).

\( \mathbf{lfp} \) and \( \mathbf{gfp} \) are the least and greatest fixpoints of

It is clear that the ascending approximation sequence AAS when starting from 1 leads to an upper bound of the least fixpoint \( \mathbf{lfp} \) of \( \widetilde{\text{Int}} \), and the truncated descending sequence TDS when starting from \( Y \) leads to an upper bound of the greatest fixpoint.
Any of the AAS, TDS, DAS, TAS methods may yields a fixpoint $fp$ which is not the fixpoint $\ellfp$ or $gfp$ of interest. None of these methods can improve $fp$ to reach $\ellfp$ or $gfp$, therefore a "fixpoint 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 $A\text{-int}$ and $B\text{-int}$ could be more refined.

Finally, going deeply into the comparison 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 Renvoise[76], Schwartz[75], Ullman[75], Wegbreit[75], ..., 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 (Sites[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.

Acknowledgments

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.


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[75]. 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.

\[
\{ (x, y) \in L^2, (x \leq y) \Rightarrow (f(x) \leq f(y)) \} 
\leq (x, y) \in L^2, \{ (f(x) \cup f(y)) \Rightarrow (x \cup f(y)) \}
\]

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

(H2): Let \( \overline{F} \) be an order-preserving function from the complete semi-lattice \( \langle L, \overline{\cup}, \overline{\leq}, \overline{\top}, \overline{\bot} \rangle \) in itself.

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

\[
\mathfrak{g} = \bigcup \{ x \mid (x \in L) \land (x \leq F(x)) \}
\]

\[
\mathfrak{l} = \bigcap \{ x \mid (x \in L) \land (F(x) \leq x) \}
\]

(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 \( \mathfrak{g} \) and \( \mathfrak{l} \) the greatest and least fixpoints of \( F \).

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

((H2.1)) \( \alpha : L \rightarrow \overline{L} \)

((H2.2)) \( \forall F \in \mathbb{F} \rightarrow F \in \mathbb{F} \).
(T1): H1, H1', H2, H3 imply that the greatest fix-points g and \( g' \) of \( F \) and \( F' \) are related by:

\( (\alpha(g) \leq \overline{g}) \) and \( (g \leq \gamma(g)) \)

Proof:

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

\[
\begin{align*}
\overline{g} & \leq \alpha(g) \leq \overline{g} & \text{trivially} \\
\overline{g} & \leq \alpha(F(g)) \leq \alpha(g) & \text{since } g = F(g) \\
\overline{g} & \leq F(\alpha(g)) \leq \alpha(g) & \text{H3.1, } \leq \text{ isotone, } \leq \text{ transitive} \\
g & \leq \overline{g} & \text{L3} \\
\gamma(\overline{g}) & \geq \gamma(\alpha(g)) & \text{H2.4} \\
\gamma(F(\overline{g})) & \geq g & \text{H2.6, } \leq \text{ transitive.}
\end{align*}
\]

Q.E.D.

Replacing \( (g, g, \overline{g}, \leq, \leq, F, F', \alpha, \gamma, \text{H3.1, } \text{H2.4, H2.6}) \) respectively by \( (g, g, \overline{g}, \leq, \leq, F, F', \alpha, \gamma, \text{H3.2, H2.3, H2.5}) \) in the above proof, we get the "dual" theorem:

(T2): H1, H1', H2, H3 imply that the least fixpoints \( \underline{g} \) and \( \underline{g'} \) of \( F \) and \( F' \) are related by:

\[
\begin{align*}
\underline{g'} & \geq \gamma(g') \geq \text{L3} \\
\gamma(\underline{g'}) & \leq \gamma(g) \leq \gamma(\overline{g}) & \text{H2.4} \\
\gamma(F(\underline{g'})) & \leq g & \text{H2.6, } \leq \text{ transitive.}
\end{align*}
\]

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

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

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

(L4): H4(H4') implies that \( F \) (\( F' \)) has a least fix-point \( \underline{x}(\overline{x}) \) which is the limit \( \cup \lim_{i \to \infty} F^i(\tau) \) of the Kleene's sequence \( 1 \leq F(\tau) \leq \ldots \leq F^n(\tau) \leq \ldots \)

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