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 Radha 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 Sintzoff[72]) is the rule of signs. The sign -1515 x 17 may be understood to denote computations on the abstract universe ((+), (-), (2)) where the semantics of arithmetic operators is defined by the rule of signs. The abstract execution -1515 x 17 \(\Rightarrow\) \((-\times \times \times \times \Rightarrow \times\)\) proves that -1515 x 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 x 17 \(\Rightarrow\) \((-\times \times \times \times \Rightarrow \times\)\)). 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:

\[ n\text{-succ}, n\text{-pred} : \text{Nodes} \rightarrow \text{Nodes} \quad \left( \, n \in \text{succ}(n) \right) \]

\[ \left(\, n \in \text{pred}(m) \right) \]

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

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

** This work was supported by IRIA-SESORT under grants 75-035 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 $\{\bot, \top\}$ to it, and imposing the ordering $\frac{1}{2} \preceq x \preceq \frac{3}{2}$ for all $x \in S$.

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

- Environments are used to hold the bindings of identifiers to their values:

  $$\text{Env} = \text{Ident}^\ast \to \text{Values}$$

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

  $$\text{val} : \text{Expr} \to [\text{Env} \to \text{Values}]$$

  In particular the projection $\text{val} \upharpoonright \text{Expr}$ is the function $\text{val} : \text{Ident} \to \text{Values}$.

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

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

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

  $$\forall s \in \text{States}, s \in (c(s), e(s))$$

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

- If $e \in \text{Env}$, $e \in \text{Values}$, $x \in \text{Ident}$ then $\text{val}[x/e] = \lambda y. \text{cond}(y = x, e, e(y))$.

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

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

  $$\text{n-state}(s) = \text{let } n \in \text{end}(c(s)), e \in \text{env}(s) \text{ within case } n \text{ in }$$

  $$\text{Assignments} \Rightarrow$$

  $$\text{Tests} \Rightarrow$$

  $$\text{Constants} \Rightarrow$$

  $$\text{Junctions} \Rightarrow$$

  $$\text{Exits} \Rightarrow$$

  $\text{esac}$

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

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

  $(\forall e \in \text{Ident}^\ast, \text{Env}(e) = \bot \text{Values})$

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

  $$\text{I-states} = \{ c(s), e \in \text{Env} \mid m \in \text{Entries} \}$$
A "computation sequence" with initial state \( i_0 \in I \)-states is the sequence:
\[
s_n = n \text{-state}^n(i_0)
\]
for \( n = 0, 1, \ldots \), where \( \text{ID} \) is the identity function and 
\( n \text{-} \text{ID} = f \preceq fn \).

The initial to final state transition function:
\[
\text{n-state} \to : \text{States} \to \text{States}
\]
is the minimal fixpoint of the functional:
\[
\lambda F. \,(n \text{-state} \circ F)
\]
Therefore:
\[
n \text{-state} \to = Y_{\text{States}} \bullet \text{States} \circ \lambda F. \,(n \text{-state} \circ F)
\]
where \( Y_{f}(f) \) denotes the least fixpoint of
\( f : D \to D \), Tarski [55].

4. Static Semantics of Programs

The constructive or operational semantics of programs defined in section 3 considers the sequence in which states occur during execution. The fundamental remark of Floyd [67] is 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 \( P \):
\[
C_q \subseteq \text{Contexts} \to = 2^{\text{Env}}
\]
\[
C_q = \{ e | \exists n \geq 0, \exists i_0 \in I \text{-states} | \langle q, e \rangle = n \text{-state}^n(i_0) \}
\]
The context vector \( CV \) associates a context to each of the program points of a program:
\[
CV \subseteq \text{Contexts} \to \text{Contexts} \to = \lambda q. \,(e | \exists n \geq 0, \exists i_0 \in I \text{-states} | \langle q, e \rangle = n \text{-state}^n(i_0))
\]
According to the semantics of programs, the context \( CV(r) \) associated to arc \( r \) is related to the contexts \( CV(q) \) at arcs \( q \) adjacent to \( r \),
\[
(\text{end}(q) = \text{origin}(r), \overrightarrow{q,r}).
\]
From the definition of the state transition function we can prove the equation:
\[
CV(r) = n \text{-context}(r, CV)
\]
where
\[
n \text{-context} : \text{Arcs} \to \text{Contexts} \to = \lambda r. \,(\text{case \, \text{origin}(r) in \, \text{Entries} \to = \{ \text{Env} \}
\]
\[
\text{Assignments} \cup \text{Tests} \cup \text{Junctions} \to = \bigcup_{e \in CV(q)} \text{env-on}(e)(n \text{-state}(\langle q, e \rangle))
\]
\[
\text{esac}
\]
We define \( \text{env-on} : \text{Arcs} \to \text{States} \to 2^{\text{Env}} \) to be
\[
\lambda r. \,(\text{as \, \text{cond}(r) = eq(s), (\text{env}(s), \emptyset))}
\]
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
\[
F \text{-cont} : \text{Contexts} \to \text{Contexts}
\]
is defined by:
\[
F \text{-cont}(CV) = \lambda r. \, n \text{-context}(r, CV)
\]
Contexts is a complete lattice with union \( \sqcup \) such that
\[
CV_1 \cup CV_2 = \lambda r. \, (CV_1(r) \sqcup CV_2(r))
\]
\( F \text{-cont} \) is order preserving for the ordering \( \sqsubseteq \) of Contexts which is defined by:
\[
(CV_1 \sqsubseteq CV_2) \iff (\forall q \in \text{Aracs}, \quad CV_1(q) \sqsubseteq CV_2(q))
\]
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 included in any solution \( \mathcal{K} \) to the system of equations:
\[
X = F \text{-cont}(X), \quad (CV \preceq \mathcal{K})
\]
Tarski [55] shows that this property uniquely determines \( CV \) as the least fixpoint of \( F \text{-cont} \). Thus \( CV \) can be equivalently defined by:
\[
D1 : CV = \lambda q. \,(e | \exists n \geq 0, \exists i_0 \in I \text{-states} | \langle q, e \rangle = n \text{-state}^n(i_0))
\]
\( \quad \text{or} \)
\[
D2 : CV = Y_{\text{Contexts}}(F \text{-cont})
\]
The concrete context vector \( CV \) is such that for any program point \( q \in \text{Arcs} \) of the program \( P \):
\[
(a) CV(q) \text{ contains at least the environments } e \text{ which may be associated to } q \text{ during any execution of } P:
\]
\[
(\exists i \geq 0, \exists i_0 \in I \text{-states} | \langle q, e \rangle = n \text{-state}^n(i_0))
\]
\[
\text{for } (e \in CV(q))
\]
\[
(b) CV(q) \text{ contains only the environments } e \text{ which may be associated to } q \text{ during an execution of } P:
\]
\[
(\exists e \in CV(q) | \exists i \geq 0, \exists i_0 \in I \text{-states} | \langle q, e \rangle = n \text{-state}^n(i_0))
\]
\( CV \) is merely a static summary of the possible executions of the program. However, our definitions \( D1 \) or \( D2 \) of \( CV \) cannot be utilized at compile time since the computation of \( CV \) 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 \( (b) \) but not necessarily \( (a) \), which therefore correctly approximate the concrete contexts we considered until now.

5. Abstract Interpretation of Programs

5.1 Formal Definition

An abstract interpretation \( \mathcal{I} \) of a program \( P \) is a tuple:
\[
\mathcal{I} = (A \text{-Cont}, \preceq, \subseteq, I, i, \text{Int})
\]
where the set of abstract contexts is a complete o-semilattice with ordering \( \preceq \), \( (x \preceq y) \iff (x + y = y) \). This implies that \( A \text{-Cont} \) has a supremum \( \top \). We suppose also \( A \text{-Cont} \) to have an infimum \( \bot \).
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 = Arcs^\0 \rightarrow A-Cont.

Whatever (C\u00b4\u00b2, C\u00b4\u00b1) \in A-Cont might be, we define:

\[ C\u00b4\u00b2 \sqsupset C\u00b4\u00b1 = \{ r \in \text{Arcs}^\0 : C\u00b4\u00b1(r) \rightarrow C\u00b4\u00b2(r) \} \]

\[ C\u00b4\u00b1 \sqsubset C\u00b4\u00b2 = \{ r \in \text{Arcs}^\0 : C\u00b4\u00b2(r) \rightarrow C\u00b4\u00b1(r) \} \]

\[ \bigtriangleup = \lambda r. T \quad \text{and} \quad \bigtriangledown = \lambda r. I \]

\[ \langle A-Cont, \sqsubseteq, \sqsupset, \bigtriangledown, \bigtriangleup \rangle \]

can be shown to be a complete lattice. The function:

\[ \text{Int} : \text{Arcs}^\0 \times A-Cont \rightarrow A-Cont \]

defines the interpretation of basic instructions. If \[ \{ C(p) \mid p \in a-prod(n) \} \]
is the set of input contexts of node \( n \), then the output context on exit arc \( r \) of \( n \) (\( r \in a-succ(n) \)) is equal to \( \text{Int}(r, C) \). Int is supposed to be order-preserving:

\[ \forall a \in \text{Arcs}, \forall (C\u00b4\u00b2, C\u00b4\u00b1) \in A-Cont', \]

\[ \{ C\u00b4\u00b2 \sqsubseteq C\u00b4\u00b1 \} \rightarrow (\text{Int}(a, C\u00b4\u00b1) \sqsubseteq \text{Int}(a, C\u00b4\u00b2)) \]

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

\[ \text{Int} : A-Cont \rightarrow A-Cont \mid \text{Int}(C\u00b4\u00b2) = \lambda r. \text{Int}(r, C\u00b4\u00b2) \]

It is easy to show that \( \text{Int} \) is order-preserving. Hence it has fixpoints, 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:

\[ C\u00b4 = \text{Int}(C\u00b4) \]

5.2 Typology of Abstract Interpretations

The restriction that "A-Cont" must be a complete semi-lattice is not drastic since Mac Kellie[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 least common upper bounds existing in S. Hence in practice the set of abstract contexts will be a lattice, which can be considered as a join (\( n \)) semi-lattice or a meet (\( n \)) semi-lattice, thus giving rise to two dual abstract interpretations.

It is a pure coincidence that in most examples (see 5.3.2) the \( n \) or \( u \) operator represents the effect of path converging. The real need for this operator is to define completeness which ensures \( \text{Int} \) to have extreme fixpoints (see 8.4).

The result of an abstract interpretation was defined as a solution to forward (\( \triangleright \)) equations: the output contexts on exit arcs of node \( n \) are defined as a function of the input contexts on entry arcs of node \( n \). One can as well consider a system of backward (\( \triangleright \)) equations: a context may be related to its successor context. The fixed points of the forward system are exactly the solutions of the backward system.

Examples:

Kildall[73] uses \( (n, \triangleright, \cdot) \), Wegbreit[75] uses \( (u, \triangleright, \cdot) \), Tenenbaum[74] uses both \( (u, \triangleright, \cdot) \) and \( (n, \triangleright, \cdot) \).

5.3 Examples

5.3.1 Static Semantics of Programs

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

\[ I_{ss} = \langle \text{Contexts}, \sqsubseteq, \text{Env}, \emptyset, n-context \rangle \]

where Contexts, \( u, \sqsubseteq, \text{Env}, \emptyset, n-context, \text{Context-Vectors}, u, \sqsubseteq, \text{F-Contexts} \) respectively correspond to A-Cont, \( \sqsubseteq, \triangleleft, \bigtriangledown, \text{Int}, A-Cont, \sqsubseteq, \triangleleft, \text{Int} \).

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.

Let Expr\(_P\) be the set of expressions occurring in a program P. Abstract contexts will be sets of available expressions, represented by boolean vectors:

\[ B\text{-vect} : \text{Expr}_P \rightarrow \{ \text{true}, \text{false} \} \]

B-vect is clearly a complete boolean lattice. The interpretation of basic nodes is defined by:

\[ \text{avail}(r, B) \]

\[ \text{let } x = \text{origin}(r) \text{ within case } n \text{ in} \]

\[ \text{Entries } \Rightarrow \lambda e. \text{false} \]

\[ \text{Assignments } \cup \text{Tests } \cup \text{Junctions } \Rightarrow \]

\[ \lambda e. (\text{generated}(n)(e) \text{ or } (\text{and } Bv(p)(e)) \text{ and } \text{transparent}(n)(e)) \]

\[ \text{esac} \]
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.

6.5.3 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 rea-
soning 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 \( \mathcal{I} = \langle \text{A-Cont}, \bar{\gamma}, \bar{x}, \bar{T}, \bar{1}, \bar{\text{Int}} \rangle \) of a program is consistent with a "concrete"
interpretation \( \mathcal{I} = \langle \text{C-Cont}, \gamma, \bar{x}, \bar{T}, \bar{1}, \bar{\text{Int}} \rangle \) if the context vector \( \bar{\gamma} \) resulting from \( \gamma \) is a cor-
rect approximation of the particular solution \( \bar{\gamma} \) result-
ing from the more refined interpretation \( \mathcal{I} \). This
may be rigorously defined by establishing a corre-
spondence (\( \alpha \) : abstraction) between concrete and ab-
stract context vectors, and inversely (\( \bar{\gamma} \) : concreti-
ization), and requiring:

\[
6.0 \quad \forall \alpha \in \text{A-Cont} \quad \bar{\gamma}(\alpha(C)) \preceq \alpha(\gamma(C)) \quad \text{and} \quad \alpha(\gamma(C)) \preceq \bar{\gamma}(\alpha(C))
\]

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

If \( f : D \rightarrow D' \) we note \( \bar{D} = \text{ArCs}^0 \circ D \) and \( \bar{D}' = \text{ArCs}^0 \circ D' \) and \( \bar{f} : \bar{D} \rightarrow \bar{D}' = \lambda (\bar{x}. f(\bar{x})) \).
We will suppose \( \alpha \) and \( \gamma \) to satisfy the following hypothesis:

\[
6.1 \quad \alpha : \text{C-Cont} \rightarrow \text{A-Cont}, \quad \gamma : \text{A-Cont} \rightarrow \text{C-Cont}
\]

6.2 \( \alpha \) and \( \gamma \) are order-preserving

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

6.4 \( \forall \bar{x} \in \text{C-Cont}, \quad x = \gamma(\alpha(x))
\]

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
\( \alpha \) is surjective and \( \gamma \) is injective. 6.4 introduces
the idea of approximation: the abstraction \( \alpha(C) \) of
a concrete context \( C \) may introduce some loss of
information so that when concretizing again \( \gamma(\alpha(c)) \)
we may get a larger context \( \gamma(\alpha(C)) \supseteq C \). Note that
it is easy to prove properties corresponding to
6.1-6.4 for \( \bar{\gamma} \) and \( \gamma \).

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:

\[
6.5 \quad \forall (a, \bar{x}) \in \text{ArCs} \times \text{A-Cont},
\gamma(\text{Int}(a, \bar{x})) \preceq \text{Int}(a, \bar{\gamma}(\bar{x}))
\]

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

\[
\begin{aligned}
\bar{C}_1 & \sim \gamma(C_1) \\
\bar{C}_0 & \sim \gamma(C_0)
\end{aligned}
\]

Suppose we want to compute the concrete output con-
text \( C_0 \) (associated with \( a \) ) resulting from con-
crete input contexts \( C_1 \) : \( C_0 = \text{Int}(a, C_1) \). We can
as well approximate this computation in the abstract
universe, and get \( C'_0 = \gamma(\text{Int}(a, \gamma(C_1))) \). 6.5 requires
\( C_0 \) to contain at least \( C_0' \), that is \( C_0 \subseteq C_0' \). On the
contrary we do not require \( C_0' \) to contain at most
\( C_0 \), that is \( C_0' \subseteq C_0 \) is not compulsory.

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

Note that \( I \leq (\alpha, \gamma) \bar{I} \) imposes a local consistency of
the interpretations \( I \) and \( \bar{I} \), 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 \( I \) and \( \bar{I} \) at the program level.

In particular if we take

\[
\mathcal{I}_{SS} = \langle \text{Contexts}, \bar{I}, \bar{\gamma}, \text{Env}, \varnothing, a-\text{context} \rangle
\]

any abstract interpretation \( \bar{I} \) of \( P \), consistent with
\( \mathcal{I}_{SS} \), is consistent with the semantic
of \( P \), which implies:

\[
\forall \bar{q} \in \text{ArCs}, \text{let} \bar{\gamma}(\bar{q}) \text{ be the result of } \bar{I},
\]

\[

\text{such that} \quad \forall I \in \text{I-states} \quad \exists-q.e^* \equiv n-\text{state}(i_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 \( P(x_1, \ldots, x_n) \)
in \( \text{Pred} \) over the program variables \( (x_1, \ldots, x_n) \in \text{Ident} \)
which are the free variables in the predicate. The
abstract interpretation is then:

\[
\mathcal{I}_{DS} = \langle \text{Pred}, \varnothing, \rightarrow, \text{true, false, n-pred} \rangle
\]
where \( n\text{-pred} \) defines Floyd[67]'s strongest post condition:

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

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

\[
\text{case } n \text{ in }
\]

\[
\text{Entries} \Rightarrow (x < \text{Ident}, x = i\text{-Values})
\]

\[
\text{Junctions} \Rightarrow \text{or (} P_v(q) \text{)}
\]

\[
\text{Tests} \Rightarrow \text{case } r \text{ in }
\]

\[
\text{esac}
\]

\[
\Rightarrow \text{a-succ}-f(n) \Rightarrow P_v(p) \text{ and not } \text{test}(n)
\]

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

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

\[
\alpha : \text{Contexts} \to \text{Pred}
\]

\[
= \lambda C . ( (C \in \text{Contexts}) \land \langle x < \text{Ident} \rangle)
\]

\[
\gamma : \text{Pred} \to \text{Contexts}
\]

\[
= \lambda P . \text{let } P \text{e}(x/y) \text{ in } x < \text{Ident}]
\]

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

\[
\{\forall x \in \text{Aracs}, \forall P_v \in \text{Pred},
\]

\[
\alpha(n\text{-context}(a, P_v)) \Rightarrow n\text{-pred}(a, P_v)
\]

See Hoare and Lauer[74], Lindley[75]. In particular Lindley[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 \( I_{BG} \leq (\alpha, \beta)I_{BG} \) has been proved, we know that the deductive semantics gives a valid proof technique, which will never permit a false theorem to be deduced:

\[
\forall q \in \text{Aracs}, \text{let } P_v \text{ be the result of } I_{BG},
\]

\[
\{n > 0, \exists i_a \in \text{I-states} \mid q, e \Rightarrow n\text{-state}^n(i_a)
\]

\[
\Rightarrow (P_v(q) \Rightarrow a(e))
\]

7. The Lattices of Abstract Interpreters

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

reflexive: \( (I \preceq (I_1, I_2)) \) where \( \lambda x . x \) is the identity function,

transitive: \( (I \preceq (\alpha_1, \gamma_1)I') \) and

\( (I' \preceq (\alpha_2, \gamma_2)I'') \) imply \( I \preceq (\alpha_1 \circ \alpha_2, \gamma_2 \circ \gamma_1)I'' \).

The relation \( \preceq \) on abstract interpretations defined by:

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

is an equivalence relation. We have:

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

The proof gives some insight in the abstraction process:

\[
1 - \{I \preceq (\beta I')\} \Rightarrow \{I \subseteq (\beta, \beta^{-1})I'\}
\]

2 - reciprocally, if \( I \preceq (\alpha_1, \gamma_1)I' \), let \( \cong \) be the relation defined on \( I \) (properly speaking, on the set of abstract contexts of \( I \) by:

\[
\{x \in (\alpha_1 y) \Rightarrow (\alpha_1 (x) \cong \alpha_1 y)\}
\]

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

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

\( \alpha_1 | \gamma_1 (I') \) and \( \alpha_2 | \gamma_2 (I) \) are bijections, hence \( \forall y' \in I' \), \( \exists ! x' \in \gamma_1 (I') \) such that \( x' = (\alpha_1 | \gamma_1 (I'))(x) \). Likewise, \( x' \in \gamma_1 (I') \) implies \( x \in I \Rightarrow \exists ! x' \in \gamma_1 (I') | \gamma_2 (I') | x = (\alpha_2 | \gamma_2 (I'))(x') \).

Therefore, \( \forall y' \in I' \), \( \exists ! x' \in \gamma_1 (I') \) such that \( x' = (\alpha_1 | \gamma_1 (I')) \circ (\alpha_2 | \gamma_2 (I'))(x) \).

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

\[
(\alpha_1 | \gamma_1 (I')) \circ (\alpha_2 | \gamma_2 (I')) \circ (\alpha_2 | \gamma_2 (I'))^{-1}
\]

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 isometric, its inverse \( \gamma_1 | \gamma_2 (I') \) is isometric and \( \gamma_1 (\text{Int}(a, x)) = \text{Int}(a, \gamma_1 (x)) \) Q.E.D.

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

In particular, we can restrict \( I \) to be set of interpretations which abstract \( I_{BG} \); \( I \) is then a lattice, (with ordering \( \preceq \)) 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 generalization 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 \( [a(S) = \text{min}(S), \text{max}(S)] \). When \( S \) is infinite the bounds will eventually be \(-\infty \) or \( +\infty \).

\( \forall f, a, b \) is \( \{x \mid a \leq x \leq b\} \). The abstract contexts...
A further abstraction may be:

$$
\alpha([a, b]) = \begin{cases} 
\text{if } a \leq b \text{ then } a & \text{if } a > 0 \\
\text{elseif } b \leq 0 & \text{else if } a \cdot b \in \mathbb{N}, \\
\gamma(0) = [0, +\infty], \quad \gamma(-) = [\infty, 0), \quad \gamma(\pm) = [\infty, +\infty]. 
\end{cases}
$$

The abstract contexts are then:

$$
I = I_I \rightarrow I_R \rightarrow I_{CP} \rightarrow I_{CS} \rightarrow I_L \rightarrow I_{SS}
$$

8. Abstract Evaluation of Programs

The system of equations:

$$
\text{Cv} : \check{\text{Int}}(\text{Cv})
$$

resulting from an interpretation $I = \langle A, \text{Cont.}, \circ, \preceq, \top, \bot, \text{Int} \rangle$ 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):

$$
\text{Cv} := (C := I; \text{until } C = \check{\text{Int}}(C) \text{ do } C := \check{\text{Int}}(C) \text{ repeat } ; C)
$$

8.1 Correctness

If $\text{Int}$ is supposed to be a complete morphism (i.e. infinitely distributive over $\circ$) then $\text{Cv}$ is the least fixpoint of $\check{\text{Int}}$, (e.g. Kildall[75]), since in a semi-lattice of finite length any distributive
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 gives through that point.

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

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

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

$$\text{Kir}(r, Cv) = \begin{cases} \text{let n be origin}(r) \text{ within} & \\
\text{case n in} & \\
\text{Entries} \Rightarrow 1 \{\text{unique entry node}\} & \\
\text{Junctions + Assignments} \Rightarrow & E \quad \text{Cv}(p) \\
\text{Tests} \Rightarrow & \\
\text{case r in} & \\
a = \text{succ}^{-1}(n) \Rightarrow & \text{Cv}(p \text{pred}(n)) \ast \frac{\text{Prob(test}(n) = \text{true})}{(1 - \text{Prob(test}(n) = \text{true}))} \\
\text{esac} & \\
\text{esac} & \\
\end{cases}$$

The main difficulty is to find the probability $\text{Prob(test}(n) = \text{true})$ 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 Kir is clearly continuous (although it is not a complete morphism) since

$$\begin{array}{c}
\text{if } \text{Cv}_0 \prec \text{Cv}_1 \prec \ldots \prec \text{Cv}_n \prec \ldots \\
\text{then } \max_{1 \leq j \leq n} \text{Cv}_j(p) \leq \text{max} \left(\frac{\text{Cv}_j(p)}{1 - \text{Prob(test}}(n) = \text{false})\right) \\
\end{array}$$

and $\max_{1 \leq j \leq n} (n_j \ast q) = (\max_{1 \leq j \leq n} n_j) \ast q$.

The least fixpoint of Kir is the limit of Kleene's sequence (the length of the sequence 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 $\infty$ limit of $0 + 1 + 1 + 1 + \ldots$
- Let $P$ be the program "begin while $T$ do $L$ 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:

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

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 $I'(1 \leq I)$ may be used for that purpose (e.g. Tenenbaum [74]). It is often better to make approximations in $I$, for example by "accelerating the convergence" of Kleene's sequences.

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

Let $I = \{\text{A-Cont}$, $\circ, \leq, 1, \tau, \text{Int}\}$ be an interpretation of $P$. When the least fixpoint $\text{CV}$ of $\text{Int}$ is unreachability, we look for an upper bound $\bar{U}$ of $\text{CV}$, since according to the correctness requirement, $\text{CV} \subseteq \gamma(\overline{\text{CV}})$ and $\text{CV} \subseteq \overline{U}$ implies $\text{CV} \subseteq \gamma(U)$.

9.1.1 Increasing Approximation Sequence

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

$$\text{A-Int}(C) = C \circ \text{Int}(C) \quad \text{and} \quad \text{not(\text{Int}(C))} \simeq C$$

$$\forall C \in (C \circ \text{Int}(C) \simeq \text{Int}(C)).$$

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

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

$$S_0 = \frac{1}{1}$$

$$S_{n+1} = \text{if } \text{not} \overline{\text{Int}(S_n)} \simeq S_n \text{ then } A-\text{Int}(S_n)$$

$$\text{else } S_n$$

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

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

Let $m$ be the least natural number (eventually infinite) such that $S_m = S_{m+1}$. For $x \in [0, m]$, we know from 9.1.1.3 that $\text{not} \overline{\text{Int}(S_k)} \simeq S_k$. Whence by definition of the ordering $\simeq$, $S_k \not\simeq \text{Int}(S_k) \simeq S_k$. Since $S_k \simeq \text{Int}(S_k) \simeq S_k$, it is always true. We can state that $S_k \simeq \overline{\text{Int}(S_k)} \simeq S_k$. Besides $\overline{\text{Int}(S_k)} \simeq S_k$ and 9.1.1.1 imply:

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

and therefore we conclude $S_{k+1} \simeq S_k$. For $x \in [1, m]$.

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 $X \in \text{A-Cont}$ such that $\overline{\text{Int}(X)} \simeq X$ (Tarski (55)) hence:

$$\forall X \in \text{A-Cont}, \overline{\text{Int}(X)} \simeq X \Rightarrow (\text{CV} \simeq X)$$

Since $S_m = S_{m+1}$ we have $\overline{\text{Int}(S_m)} \simeq S_m$ and therefore $\overline{\text{Int}(S_m)} \simeq S_m$ is a correct approximation of $\overline{\text{Int}(X)}$. 245
Let us note \([a, b]\) where \(a \leq x \leq b\) the predicate. The system of equations corresponding to the example is:

(0) \(C_0 = [0, 0]\)
(1) \(C_1 = [1, 1]\)
(2) \(C_2 = C_0 \sqcup C_1\)
(3) \(C_3 = C_2 \sqcap [1, 100]\)
(4) \(C_4 = C_3 \sqcup [1, 1]\)
(5) \(C_5 = [2, 101]\)

Note: \(C_1 \sqcup C_4 = [1, 101] \leq C_2 = [1, +\infty]\) stop on that path.

\(C_5 = C_2 \cap [101, +\infty]\)
\(= [1, +\infty] \cap [101, +\infty]\)

\(\star C_5 = [101, +\infty]\)

exit, stop.

The final context on each arc is marked by a star \(\star\). Note that the results are approximate ones, e.g., \(C_5\).

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

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

$$9.3.2.1 \quad \{ W \in A\text{-Cont} \} \implies \{ C \supseteq \overline{\text{D-int}}(C) \supseteq \text{Int}(C) \}$$

$$9.3.2.2 \quad \forall W \in A\text{-Cont}^2, \text{ every infinite sequence } C_n, \overline{\text{D-int}}(C_n), \ldots, \overline{\text{D-int}}^n(C_n), \ldots \text{ is not strictly decreasing.}$$

The truncated decreasing sequence $S_n^0, \ldots, S_n^n$ is recursively defined by:

$$9.3.2.3 \quad S_n^0 = S_n \quad S_n^{n+1} = \begin{cases} \text{if } (S_n^i \not\supseteq \text{Int}(S_n^i)) \text{ and } (S_n^i \not\supseteq \overline{\text{D-int}}(S_n^i)) & \text{then } \overline{\text{D-int}}(S_n^i) \\ \text{else} & S_n^i \end{cases}$$

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

Let $p$ be the least natural number (eventually infinite) such that $S_p^i = S_p^{p+1}$. Trivially from 9.1.1:

$$S_0^0 - S_n \supseteq \text{Int}(S_n^0) \supseteq CV$$

If $p > 0$ then $S_0^0 \not\supseteq \text{Int}(S_0^0)$, therefore $S_0^0 \supseteq \text{Int}(S_0^0)$. Then applying 9.3.2.1 we have:

$$S_0^0 \supseteq \overline{\text{D-int}}(S_0^0) = S_0^1 \supseteq \text{Int}(S_0^0) \supseteq CV$$

But 9.3.2.3 implies $S_0^0 \not\supseteq \text{Int}(S_0^0)$, hence:

$$S_0^0 > S_0^1 \supseteq \text{Int}(S_0^0) \supseteq CV$$

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

$$S_{k+1}^1 - S_k^1 \supseteq \text{Int}(S_k^1) \supseteq CV$$

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

$$\text{Int}(S_{k+1}^1) \supseteq \text{Int}(S_k^1) \supseteq \text{Int}^2(S_{k+1}^1) \supseteq \text{Int}(CV)$$

By transitivity $S_{k+1}^1 \supseteq \text{Int}(S_{k+1}^1)$ and since 9.3.2.3 implies $S_k^1 \not\supseteq \text{Int}(S_k^1)$ we have from 9.3.2.1:

$$S_k^1 \supseteq \overline{\text{D-int}}(S_k^1) = S_{k+1}^1 \supseteq \text{Int}(S_k^1)$$

Since 9.3.2.3 implies $S_k^1 \not\supseteq \overline{\text{D-int}}(S_k^1)$ we have:

$$S_{k+1}^1 \supseteq S_{k+1}^1 \supseteq \text{Int}(S_k^1) \supseteq CV$$

By recurrence on $k$ the result is true for $k < p$. Moreover 9.3.2.2 implies that $p$ is finite. Q.E.D.

9.3.3 Generalization of Kleene's Descending Sequence

When $A\text{-Cont}$ satisfies the descending chain condition, one can choose $\overline{\text{D-int}}$ to be $\text{Int}$, in which case the final result $S_0^p = \text{Int}(p)$ is a fixpoint greater or equal to the least fixpoint $CV$ of $\text{Int}$.

The limit of the descending sequence $S_0^0 = \tilde{S}_0, \ldots, S_0^n, \ldots$ 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 $\overline{\text{D-int}}$ by local modifications to $\text{Int}$:

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

$$9.3.4.2 \quad \forall W(C, C' \subseteq A\text{-Cont}^2, \quad (C \supseteq C') \implies (C \supseteq C \land C' \supseteq C')$$

9.3.4.3 Every infinite sequence $S_0, \ldots, S_n$ of the form $S_0^0 = C_0, S_1^1 = S_0 \land C_1, \ldots, S_n^i = S_{i-1} \land C_i$, ... for arbitrary abstract contexts $C_0, C_1, \ldots, C_n$, ... is not strictly decreasing.

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

$$9.3.4.4 \quad \overline{\text{D-int}} = \lambda(q, CV). \text{ if } q \in W\text{-arcs then } CV(q) \lor \text{Int}(q, CV)$$

$$\text{else } \text{Int}(q, CV)$$

This definition of $\overline{\text{D-int}}$ trivially satisfies the requirement 9.3.2.1 since $CV \supseteq \text{Int}(CV)$ implies $CV(q) \supseteq \text{Int}(q, CV)$. Let $W \in \text{Arcs}$. If $q \in W\text{-arcs}$ then 9.3.4.2 implies that $CV(q) \supseteq CV(q) \lor \text{Int}(q, CV) = \overline{\text{D-int}}(q, CV)$. Otherwise, if $q \not\in W\text{-arcs}$ $CV(q) \supseteq \text{Int}(q, CV) = \text{D-int}(q, CV)$. Hence $CV \supseteq \overline{\text{D-int}}(CV) \supseteq CV$.

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

9.4 Example : Bounds of Integer Variables

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

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

The ascending approximation sequence led to the approximate solution:

$$C_1 = [1, 1]$$  
$$C_2 = [1, 10]$$  
$$C_3 = [1, 100]$$  
$$C_4 = [2, 101]$$  
$$C_5 = [101, +\infty]$$

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

$$[i, j] \Delta [k, \ell] = \begin{cases} \text{if } i = -\infty \text{ then } k \text{ else } \min(i, k), j \text{ if } j = +\infty \text{ then } k \text{ else } \max(i, k), \ell \end{cases}$$

248
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) $C_2 = C_2 \cup (C_1 \cup C_4)$

The descending approximation sequence is:

- $C_2 = C_2 \cup (C_1 \cup C_4) = [1, +\infty) \cup ([1, 1] \cup [2, 101])$
- $C_3 = C_2 \cap [-\infty, 100]$
- $C_4 = C_3 \cap [101, +\infty]$
- $C_5 = [1, 101] \cap [101, +\infty] = [101, 101]$

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 $\mathcal{A}$-cont may be partitioned as follows:

$\mathcal{A}$f is the least and greatest fixpoints of $\mathcal{A}$-cont. The ascending (AKS) and descending (DRS)
Kleene's sequences converge toward $\mathcal{A}$f and gfp respectively. These limits are reached when $\mathcal{A}$-cont is continuous. When AKS is infinite we have proposed to use an ascending approximation sequence (AAS) to approximate $\mathcal{A}$f. Its limit may be some fixpoint $\hat{\mathcal{A}}$p that $\mathcal{S}_m \supset \mathcal{A}$-cont(s_m) and $\mathcal{A}$f leads to an upper bound of the least fixpoint $\mathcal{A}$f of $\mathcal{A}$-cont, 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 $\mathcal{A}$-cont.
Any of the AAS, TDS, DAS, TAS methods may yields a fixpoint \( fp \) which is not the fixpoint \( ffp \) or \( gfp \) of interest. None of these methods can improve \( fp \) to reach \( ffp \) or \( gfp \), therefore a "fix-point improvement method" is necessary. It is our feeling that such a method could be designed only when considering that A-Gont possesses a richer structure (i.e. for particular applications).

Furthermore, in the AAS, TDS, DAS, TAS sequences the term of rank \( n \) is computed only as a function of the term of rank \( n-1 \), hence these are "separate steps" methods. One can as well imagine to use "bound steps" methods, where the term of rank \( n \) is computed as a function of the terms of rank \( n-1, n-2, \ldots, n-k \). 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 comparism with numerical analysis methods, it is clear that some measure is necessary to control the accuracy of the result. Its definition would certainly also necessitate some additional properties of the abstract contexts.

10. Conclusion

It is our feeling that most program analysis techniques may be understood as abstract interpretations of programs. Let us point out global data flow analysis in optimizing compilers (Kildal[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 built 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.


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


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


13. Appendix

We note \( L, \leq, \leq, \leq, \leq, \leq \) a complete \( \leq \)-semilattice \( L \), with partial ordering \( \leq \), supremum \( \leq \) and infimum \( \leq \). 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:

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

(1H): Let \( f \) be an order-preserving function from the complete semi-lattice \( L, \leq, \leq, \leq, \leq, \leq \) in itself.

(1H): Let \( \bar{f} \) be an order-preserving function from the complete semi-lattice \( L, \leq, \leq, \leq, \leq \) in itself.

(1L): The fixpoints of \( f \) form a non-empty complete lattice with supremum \( g \), infimum \( k \) such that:

\[ g = \alpha \leq \forall x \in L, \leq (x \leq f(x)) \]

\[ k = \alpha \leq \forall x \in L, \leq (f(x) \leq x) \]

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

We note \( g \) and \( \bar{f} \) the greatest and least fixpoints of \( f \).

(2H): Let \( \alpha \) and \( \beta \) be such that:

\( \forall (x) \in L, \leq (f(x) = \alpha(x)) \)

\( \forall (x) \in L, \leq (g(x) = \beta(x)) \)

(3H): \( (1H), (1H), (2H) \) and \( \forall x \in L, \leq (\forall (x) \leq \forall (x)) \)

(3H): \( (1H), (1H), (2H) \) and \( \forall x \in L, \leq (\forall (x) \leq \forall (x)) \)

\( \forall x \in L, \leq (\forall (x) \leq \forall (x)) \)

(3L): \( [H.1] \leftrightarrow [H.2] \)

\( \forall x \leq \forall (x), \leq (\forall (x) = \alpha(x)) \in \forall (x) \in \forall (x) \)

\( \forall (x), \leq (\forall (x) = \alpha(x)) \in \forall (x) \in \forall (x) \)

\( \forall (x), \leq (\forall (x) = \alpha(x)) \in \forall (x) \in \forall (x) \)

\( \forall (x), \leq (\forall (x) = \alpha(x)) \in \forall (x) \in \forall (x) \)

\( \forall (x), \leq (\forall (x) = \alpha(x)) \in \forall (x) \in \forall (x) \)

Since \( H.1 \) and \( H.2 \) are proved by \( L.2 \) to be equivalent, we choose:

(3H): \( [H.1] \) or \( [H.2] \)

(3L): Let \( F: L \rightarrow L \) be an order-preserving function from the semilattice \( L, \leq, \leq, \leq, \leq, \leq \) in itself, \( \leq \) and \( \leq \) respectively the least and greatest fixpoints of \( F \), then:

\( \forall x \in L, \{g \leq F(x) \geq x\} \leftrightarrow \{g \geq x\} \)

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

\( \forall x \in L, \{\leq \leq F(x) \geq x\} \leftrightarrow \{\leq \leq x\} \)

\( \forall x \in L, \{\leq \leq F(x) \geq x\} \leftrightarrow \{\leq \leq x\} \)

251
(T1): \( H_1, H_1, H_2, H_3 \) imply that the greatest fixpoints \( g \) and \( \bar{g} \) of \( F \) and \( \bar{F} \) are related by:

\[
(a(g) \preceq \bar{g}) \quad \text{and} \quad (\bar{g} \preceq \gamma(\bar{g}))
\]

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

\[
\begin{align*}
\bar{g} & \preceq \alpha(g) \quad \text{trivially} \\
\bar{g} & \preceq \alpha(F(g)) \preceq \alpha(g) \quad \text{since } g = F(g) \\
\bar{g} & \preceq \alpha(\bar{F}(g)) \preceq \alpha(g) \quad H3.1. \cup \text{ isotone}, \preceq \text{ transitive} \\
\bar{g} & \preceq \alpha(g) \quad L3 \\
\gamma(\bar{g}) & \succeq \gamma(\alpha(g)) \quad H2.4 \\
\gamma(F) & \preceq g \quad H2.6, \preceq \text{ transitive.}
\end{align*}
\]

Q.E.D.

Replacing \(<g, \bar{g}, \cup, \preceq, \geq, F, \bar{F}, \alpha, \gamma, H3.1, H2.4, H2.6>\) respectively by \(<L, \cup, \preceq, \geq, F, \bar{F}, \alpha, \gamma, H3.1, H2.4, H2.6, H2.3, H2.5>\) in the above proof, we get the "dual" theorem:

(T2): \( H_1, H_1, H_2, H_3 \) imply that the least fixpoints \( L \) and \( \underbar{L} \) of \( F \) and \( \bar{F} \) are related by:

\[
(\gamma(L) \succeq L) \quad \text{and} \quad (L \preceq \alpha(L))
\]

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

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

(H\#4): Let \( \bar{F} \) be a continuous function from the complete semi-lattice \(<L, \cup, \preceq, \tau, \iota>\) in itself.

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

(L4): H4(H\#4) implies that \( F(\bar{F}) \) has a least fixpoint \( \iota(\bar{F}) \) which is the limit \( \bigcup_{i=0}^{\infty} F^i(\iota) \) of the Kleene's sequence \( 1 \leq F(1) \leq \ldots \leq F^n(1) \leq \ldots \)

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