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

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

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

Abstract program properties are modeled by a complete semilattice, Birkhoff[61]. Elementary pro-
An assignment node \((n \in \text{Assignments})\) has one predecessor and one successor \((|\text{pred}(n)| = 1\) and \(|\text{succ}(n)| = 1\)). Let "Ident" and "Expr" be the distinct syntactic categories of identifiers and expressions. An assignment node \(n\) assigns the value of the right hand-side expression \(\text{expr}(n)\) to the left hand-side identifier \(\text{id}(n)\):

\[
\begin{align*}
\text{expr} & : \text{Assignments} \rightarrow \text{Expr} \\
\text{id} & : \text{Assignments} \rightarrow \text{Ident}
\end{align*}
\]

A test node \((n \in \text{Tests})\) has a predecessor and two successors, \((|\text{pred}(n)| = 1\) and \(|\text{succ}(n)| = 2\)). The true and false successor nodes are respectively denoted \(\text{succ}-t(n)\) and \(\text{succ}-f(n)\):

\[
\begin{align*}
\text{test} & : \text{Tests} \rightarrow \text{Bool} \\
\text{true} & : \text{Tests} \rightarrow \text{Bool} \\
\text{false} & : \text{Tests} \rightarrow \text{Bool}
\end{align*}
\]

Let "Bexp" be the syntactic category of boolean expressions, each test node \(n\) contains a boolean expression \(\text{test}(n)\):

\[
\begin{align*}
\text{test} & : \text{Tests} \rightarrow \text{Bexp} \\
\text{true} & : \text{Tests} \rightarrow \text{Bexp} \\
\text{false} & : \text{Tests} \rightarrow \text{Bexp}
\end{align*}
\]

A junction node \((n \in \text{Junctions})\) has one successor and more than one predecessor, \((|\text{pred}(n)| > 1\) and \(|\text{pred}(n)| > 1\)). Immediate predecessor nodes of a junction node are not junction nodes, \((\forall m \in \text{Junctions}, \forall n \in \text{pred}(n), \text{not}(m \in \text{Junctions}))\).

An exit node \(n\) has one predecessor and no successors, \((|\text{pred}(n)| = 1\) and \(|\text{succ}(n)| = 0\)).

The set "Arcs" of edges of a program is a subset of \(\text{Nodes} \times \text{Nodes}\) defined by:

\[
\text{Arcs} = \{(n,m) | (n \in \text{Nodes}) \text{ and } (m \in \text{succ}(n))\}
\]

which may be equivalently defined by:

\[
\text{Arcs} = \{(n,m) | (n \in \text{Nodes}) \text{ and } (m \in \text{pred}(n))\}
\]

We will assume that the directed graph \(\text{Nodes, Arcs}\) is connected.

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 \(\bot, \top\) to it, and imposing the ordering \(\bot \leq x \leq \top\) 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 \(\text{expr} \in \text{Expr}\) in the environment \(e \in \text{Env}\) is given by \(\text{val} \equiv \text{expr}(e)\) so that:
    \[
    \text{val} : \text{Expr} \rightarrow \text{Env} \rightarrow \text{Values}
    \]
  - In particular the projection \(\text{val} \equiv \text{Expr}\) of the function \(\text{val} \equiv \text{Expr}\) in domain \(\text{Expr}\) has the functionality:
    \[
    \text{val} : \text{Expr} \rightarrow \text{Env} \rightarrow \text{Booll}
    \]
- The state set "States" consists of the set of all information configurations that can occur during computations:
  \[
  \text{States} = \text{Arcs}^* \times \text{Env}
  \]
A state \((s \in \text{States})\) consists of a control state \(\text{cs}(s)\) and an environment \(\text{env}(s)\), such that:
  \[
  \forall s \in \text{States}, \ s = <\text{cs}(s), \text{env}(s)>.
  \]
- We use a continuous conditional function \(\text{cond}(b, e_1, e_2)\) equal to \(b, e_1, e_2\) or \(T\) respectively as the value of \(b\) is \(0, \text{true}, \text{false}\) or \(T\). We also use \(\text{if } b \text{ then } e_1 \text{ else } e_2 \text{ if } b\) to denote \(\text{cond}(b, e_1, e_2)\).
- If \(e \in \text{Env}, v \in \text{Values}, x \in \text{Ident} \) then:
  \[
  
  \]
A "computation sequence" with initial state. Since the equation $Cy(r) = n-context(r, Cy)$ must

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 × A-Cont^2. Whatever (Cv', Cv'') ∈ A-Cont^2 may be, we define:

\[ \text{Cv'} \bowtie \text{Cv''} = \langle \lambda r. \text{Cv'}(r) \circ \text{Cv''}(r) \rangle \]

\[ \tilde{\tau} = \lambda r. \tau \text{ and } \tilde{1} = \lambda r. 1 \]

\(<\text{A-Cont}, \bowtie, \subseteq, \tilde{\tau}, \tilde{1} \rangle \) can be shown to be a complete lattice. The function:

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

defines the interpretation of basic instructions. If \( \{ C(q) \} q \in a-prod(n) \) 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 \( \text{Int}(r, C) \). \( \text{Int} \) is supposed to be order-preserving:

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

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

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

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

\[ \text{Cv} = \tilde{\text{Int}}(\text{Cv}) \]

5.2 Typology of Abstract Interpreters

The restriction that "A-Cont" must be a complete semi-lattice is not drastic since Mac Nile[37] showed that any partly ordered set \( S \) can be embedded in a complete lattice so that inclusion is preserved, together with all greatest lower bounds and lowest upper bounds existing in \( S \). Hence in practice the set of abstract contexts will be a lattice, which can be considered as a join (\( u \)) 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 in defining operators which ensure that when

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

\[ \text{Exp}(r, \text{bv}) \text{ within } \]

Examples:

Kildall[73] uses \( (n, \rightarrow, \tau) \), Wegbreit[75] uses \( (u, \rightarrow, i) \). Tenenbaum[74] uses both \( (u, \rightarrow, i) \) and \( (n, \tau, i) \).

5.3 Examples

5.3.1 Static Semantics of Programs

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

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

where Contexts, \( u, \subseteq, \text{Env}, \emptyset, n-context \), Context-Vectors, \( u, \subseteq, \text{F-Context} \) respectively correspond to \( \text{A-Cont}, u, \subseteq, \tilde{\tau}, \tilde{1}, \tilde{\text{Int}}, \text{A-Cont}, u, \subseteq, \tilde{\text{Int}} \).

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

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

\[ \text{B-vec} \text{ is clearly a complete boolean lattice. The interpretation of basic nodes is defined by :} \]

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

\[ \text{Exp}(r, \text{bv}) \text{ within } \]
The determination of available expressions, back-
dominators, intervals, ... requires a forward sys-
tem of equations. Some global flow problems, nota-
ibly the live variables and very busy expressions
require nonmonotone information backward through
Instead of the global hypothesis 6.0 we will use
the following local hypothesis on the concrete and
abstract interpretations of primitive language cons-
structs:

\[ \cdots = \_ \_ \_ \_ \_ \_ \]
where \texttt{n-pred} defines Floyd\textasciitilde s strongest post condition:

\[
\texttt{n-pred}(r, \texttt{Pv}) = \\
\text{let}(n \text{ be origin}(r), \langle p \text{ be } a\text{-pred(origin}(r))\rangle \text{within} \\
\text{case } n \text{ in} \\
\begin{align*}
\text{Entries} & \Rightarrow (\forall x \in \text{Ident}, x = ^i\text{Values}) \\
\text{Junctions} & \Rightarrow \texttt{or}(\texttt{Pv}(q)) \\
\text{Tests} & \Rightarrow \texttt{case } r \text{ in} \\
& \quad (a\text{-succ}(n)) \Rightarrow \texttt{Pv}(p) \text{ and } \texttt{test}(n) \\
& \quad (a\text{-succ}(n)) \Rightarrow \texttt{Pv}(p) \text{ and } \texttt{not test}(n) \\
\text{esac}
\end{align*}
\]

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

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

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

is an equivalence relation. We have:

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

The proof gives some insight in the abstraction process:

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

2 - reciprocally,

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

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

\(x \in I', \gamma(x)\) each equivalence class \(C_\gamma = \{x \in I \mid \alpha_1(x) = x'\}\) has a least upper bound \(\gamma(x')\) 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 representers of the equivalence classes on \(I\).

Let us show now that under the hypothesis
8. Abstract Evaluation of Programs

The system of equations:

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

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

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

8.1 Correctness

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

8.2 Termination

The abstract evaluation terminates iff Kleene's sequence is finite. This may be the case because \( A\text{-Cont} \) is finite (e.g., type checking in ALGOL 60, Nauf[65]), or a finite subset only is to be considered for any particular program (e.g., type checking in ALGOL 68), or \( A\text{-Cont} \) may be of finite length \( m \) (the length of any strictly increasing chain is bounded by \( m \), Kildall[73], Wegbreit[75]) or \( A\text{-Cont} \) may satisfy the ascending chain condition (every strictly increasing chain is finite, although not bounded). A lattice may have infinite chains, although \( \text{Int} \) is chosen so that Kleene's sequences are finite. Finally an infinite Kleene's sequence may be arbitrarily truncated (to get a lower bound of its limit), some induction principle (Sintzoff [75]) or heuristics (Katz and Manual[76]) may be used to pass to the limit, or approximate it, (Coates[76]).

8.3 Efficiency

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

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

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

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

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

$$\text{Kir}(r, Cv) = \begin{cases} \text{let } n \text{ be origin}(r) \text{ within} \\ \text{case } r \text{ in} \\ \text{Entries} \Rightarrow 1 \text{ (unique entry node)} \\ \text{Junctions} \cup \text{Assignments} \Rightarrow \bigwedge_{p \in A\text{-pred}(n)} \text{Cv}(p) \\ \text{Tests} \Rightarrow \\ \text{case } r \text{ in} \\ \{a\rightarrow \neg r(n)\} \Rightarrow \text{Cv}(\text{a-pred}(n)) \ast \frac{1}{\text{Prob}(\neg r(n) = \text{true})} \\ \{a\rightarrow \neg r(n)\} \Rightarrow \text{Cv}(\text{a-pred}(n)) \ast \frac{1}{\text{Prob}(\neg r(n) = \text{true})} \\ \text{ esac} \end{cases}$$

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

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

$$\text{if } \text{Cv}_0 \preceq \text{Cv}_1 \preceq \ldots \preceq \text{Cv}_\infty \preceq \ldots$$

then

$$\operatorname{max}(\bigwedge_{p \in A\text{-pred}(n)} \text{Cv}_i(p)) = \bigwedge_{p \in A\text{-pred}(n)} \operatorname{max}(\text{Cv}_i)(p)$$

and

$$\max(n_1 \ast q) = (\max(n_1)) \ast q$$

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

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

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 $I'$ (where $I'$ is not required for that purpose (e.g., Tarski[1955])) is often better to make approximations in $I$, for example by "accelerating the convergence" of Kleene’s sequence.

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

Let $I = \langle A\text{-Cont}, \ast, \leq, 1, \top, \text{Int} \rangle$ be an interpretation of $P$. When the least fixpoint $\text{Cv}$ of $\text{Int}$ is unreachable, we look for an upper bound $\text{Ub}$ of $\text{Cv}$, since according to the correctness requirement $6.0$, $\text{Cv} \preceq \gamma(\text{Cv})$ and $\text{Cv} \preceq \text{Ub}$ implies $\text{Cv} \preceq \gamma(\text{Ub})$.

9.1.1 Increasing Approximation Sequence

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

9.1.1.1 $(\forall n \geq 0, C = \text{A-Int}(n) \ast C$ and not $(\text{Int}(C) \leq C))$]

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

The approximation sequence $S_0, S_1, \ldots$ is recursively defined by:

9.1.1.3 $S_0 = I$

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

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

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

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

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

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

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

Let $\text{Cv}$ be the least fixpoint of $\text{Int}$, it is the greatest lower bound of the set of $X \in \text{A-Cont}$ such that $(\text{Int}(X) \preceq X$ (Tarski[1955]) hence:

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

Since $S_m = S_{m+1}$ we have $(\text{Int}(S_m) \preceq S_m$ and therefore $\text{Cv} \preceq S_m$. $S_m$ is a correct approximation of $\text{Cv}$. 

---

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

\[
\exists C \in \text{Arcs} \mid C \in \text{A-Cont} \land \text{Int}(q, C) \neq \text{Int}(q, C[r/C])
\]

As before, we define:

9.1.3.5 \(A\text{-Int} = \lambda C \psi \cdot (\lambda q. A\text{-int}(q, C\psi))\)

Now we have to show that this definition of A-int satisfies the requirements 9.1.1.2 and 9.1.1.7.

Let us consider a sequence \(S_0 = \bar{\bar{1}}, \ldots, S_{n+1} = A\text{-int}(S_n)\), ... We show that this sequence is increasing that is to say:

9.1.3.6 \(S_n \preceq A\text{-int}(S_n), \forall n \geq 0\).

Trivially for \(n = 0\), \(S_0 = \bar{\bar{1}} \preceq A\text{-int}(S_0)\). For the induction step, suppose the result to be true for \(n \leq m\). Let us prove that:

\(S_{n+1} \preceq A\text{-int}(S_{n+1})\)

\(\iff S_{n+1}(q) \preceq A\text{-int}(q, S_{n+1}), \forall q \in \text{Arcs}.

If \(q \in W\text{-arcs}, \text{then } A\text{-int}(q, S_{n+1}) = \)

\(...) \psi \Gamma \text{Int}(\sigma, S) \supset S_{n+1}(\psi) \Gamma \text{Int}(\sigma, S_{n+1}) \ldots \)
\* C3 = [1, 100]  
C4 = C3 \* [1, 1]  
= [1, 100] \* [1, 1]  
* C4 = [2, 101]  
Note : C1 \* C4 = [1, 101] \leq C2 = [1, +\infty]  
stop on that path.  
C5 = C2 \cap [101, +\infty]  
= [1, +\infty] \cap [101, +\infty]  
* C5 = [101, +\infty]  
exit, stop.

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 like filters. Furthermore, for PASCAL like...
(However, we will not artificially truncate the decreasing sequence by imposing an arbitrary upper bound on \( n \)).

Let \( \text{D-int} : A\text{-Cont} \rightarrow A\text{-Cont} \) be such that:

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

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 \( S'_1, \ldots, S'_n, \ldots \) is recursively defined by:

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

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

Let \( p \) be the least natural number (eventually infinite) such that \( S'_p = S'_{p+1} \). Trivially from 9.1.1:
\[
S'_0 = S_m \supseteq \text{Int}(S'_0) \supseteq \text{CV}
\]

If \( p > 0 \) then \( S'_0 \neq \text{Int}(S'_0) \), therefore \( S'_0 \supseteq \text{Int}(S'_0) \).

Then applying 9.1.2.1 we have:
\[
S'_1 \supseteq \text{D-int}(S'_0) = S'_1 \supseteq \text{Int}(S'_0) \supseteq \text{CV}
\]

But 9.3.2.3 implies \( S'_1 \neq \text{D-int}(S'_1) \), hence:
\[
S'_0 \supsetneq S'_1 \supseteq \text{Int}(S'_0) \supseteq \text{CV}
\]

For the induction step, let us suppose that for \( S'_p \):
\[
S'_p \supsetneq S'_{p+1} \supseteq \text{Int}(S'_p) \supseteq \text{CV}
\]

\[ \text{D-int} : \text{Arcs} \times A\text{-Cont} \rightarrow A\text{-Cont} \] is defined by:

9.3.4.4 \( \text{D-int} = \lambda(q, \text{CV}) . \begin{cases} \text{if } q \in \text{W-arcs} \text{ then } \\ \text{CV}(q) \supset \text{Int}(q, \text{CV}) \end{cases} \)

This definition of \( \text{D-int} \) trivially satisfies the requirement 9.3.2.1 since \( \forall q \in \text{Arcs} \) with property \( \text{CV} \supset \text{Int}(\text{CV}) \) implies \( \text{CV}(q) \supset \text{Int}(q, \text{CV}) \).

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

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

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 approximation interpretation
\[ \text{D-int} : \text{Arcs} \times A\text{-Cont} \rightarrow A\text{-Cont} \] is defined by:

9.3.4.4 \( \text{D-int} = \lambda(q, \text{CV}) . \begin{cases} \text{if } q \in \text{W-arcs} \text{ then } \\ \text{CV}(q) \supset \text{Int}(q, \text{CV}) \end{cases} \)

This definition of \( \text{D-int} \) trivially satisfies the requirement 9.3.2.1 since \( \forall q \in \text{Arcs} \) with property \( \text{CV} \supset \text{Int}(\text{CV}) \) implies \( \text{CV}(q) \supset \text{Int}(q, \text{CV}) \).

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

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

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 approximation interpretation
\[ \text{D-int} : \text{Arcs} \times A\text{-Cont} \rightarrow A\text{-Cont} \] is defined by:

9.3.4.4 \( \text{D-int} = \lambda(q, \text{CV}) . \begin{cases} \text{if } q \in \text{W-arcs} \text{ then } \\ \text{CV}(q) \supset \text{Int}(q, \text{CV}) \end{cases} \)

This definition of \( \text{D-int} \) trivially satisfies the requirement 9.3.2.1 since \( \forall q \in \text{Arcs} \) with property \( \text{CV} \supset \text{Int}(\text{CV}) \) implies \( \text{CV}(q) \supset \text{Int}(q, \text{CV}) \).

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

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

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.
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:

- \( C_2 = C_2 \triangle (C_1 \cup C_4) \)
- \( = [1, +\infty) \triangle ([1, 1] \cup [2, 101]) \)
- \( = [1, +\infty) \triangle [1, 101] \)
- \( C_2 = [1, 101] \)
- \( C_3 = C_2 \cap [-\infty, 100] \)
- \( C_3 = [1, 101] \cap [-\infty, 100] = [1, 100] \)
- Stop on this 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

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 \( S_0 \) remain in the partition \( \{ X \mid X \geq \text{Int}(X) \} \), so that their limit \( S^\star \) is greater than \( \text{gfp} \).
Any of the AAS, TDS, DAS, TAS methods may yields a fixpoint fp which is not the fixpoint lfp or gfp of interest. None of these methods can improve fp to reach lfp 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 case the Kleene's sequences may be used to compute the first k terms. After k steps more information about the program would be available to heuristically accelerate the convergence so that the definition of A-int and D-int could be more refined.

Finally, going deeply into the 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 (Kildal [73], Morel and Renvoize [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.

Acknowledgements

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

11. References


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


Hewitt et al. [73]. Actor induction and meta-evaluation. Conf. Rec. of the First ACM Symp. on Principles of Programming Languages, Boston, 153-168, (Oct.).


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

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


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


Naur [65]. Checking of operand types in ALGOL compilers, BIT 5, 151-163.
\[ \begin{align*}
\forall (x, y) \in X, \quad (x \leq y) & \iff \{x \leq y\} \\
\text{Let } P \text{ be an order-preserving function from } \text{the complete semi-lattice } \mathcal{L}, \{', \leq\} \text{ to itself.} \\
\text{Let } g \text{ be an order-preserving function from } \{', \leq\} \text{ to itself.} \\
\text{The fsiprines of } P \text{ form a congruence relation, } \\
\mathcal{L} = \{x \in X \mid (x, P(x)) \in (x, g(x))\} \\
\delta = \{(x, P(x)) \mid (x, g(x)) \in \mathcal{L}\} \\
\{x \in X \mid (x, P(x)) \in \mathcal{L}\} \\
\end{align*} \]
(T1): H1, H2, H3 imply that the greatest fix-
points \( \varphi \) and \( \gamma \) of \( \mathcal{F} \) and \( \mathcal{F} \) are related by:

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

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

\[
\begin{align*}
\tilde{g} & \sqsubseteq \alpha(g) \quad \text{trivially} \\
\tilde{g} & \sqsubseteq \alpha(F(g)) \quad \text{since } g = F(g) \\
\tilde{g} & \sqsubseteq \alpha(F(\alpha(g))) \quad \text{H3.1, } \cup \text{ isotone, } \sqsubseteq \text{ transitive} \\
\tilde{g} & \sqsubseteq \alpha(g) \quad \text{L3} \\
\gamma(\tilde{g}) & \geq \gamma(\alpha(g)) \quad \text{H2.4} \\
\gamma(\tilde{g}) & \geq g \quad \text{H2.6, } \geq \text{ transitive.}
\end{align*}
\]
Q.E.D.

Replacing \( \langle g, \tilde{g}, \sqsubseteq, \geq, F, \tilde{F}, \alpha, \gamma, H3.1, H2.4, H2.6 \rangle \) respectively by \( \langle \mathcal{F}, \mathcal{F}, \sqsubseteq, \geq, \mathcal{F}, \tilde{F}, \alpha, \gamma, H3.1, H2.4, H2.6 \rangle \).