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

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

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

A program denotes computations in some universe of objects. Abstract interpretation of programs consists in using that denotation to describe computations in another universe of abstract objects, so that the results of abstract execution give some informations on the actual computations. An intuitive example (which we borrow from Sintoff [72]) is the rule of signs. The text -1515 * 17 may be understood to denote computations on the abstract universe \((+), (-), (\emptyset)\) where the semantics of arithmetic operators is defined by the rule of signs. The abstract execution \(-1515 * 17 \Rightarrow -(+) * (+) \Rightarrow -(+) * (+) \Rightarrow (-),\) proves that \(-1515 * 17\) is a negative number. Abstract interpretation is concerned by a particular underlying structure of the usual universe of computations (the sign, in our example). It gives a summary of some facets of the actual executions of a program. In general this summary is simple to obtain but inaccurate (e.g. \(-1515 + 17 \Rightarrow -(+) + (+) \Rightarrow (-) + (+) \Rightarrow (\emptyset)).\) Despite its fundamentally incomplete results abstract interpretation allows the programmer or the compiler to answer questions which do not need full knowledge of program executions or which tolerate an imprecise answer, (e.g. partial correctness proofs of programs ignoring the termination problems, type checking, program optimizations which are not carried in the absence of certainty about their feasibility, ...).

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 built 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.

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.

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:

\[
\begin{align*}
\text{n-succ, n-pred : Nodes } & \rightarrow \text{Nodes} \\
& (m \in \text{n-succ}(n)) \\
& \iff (n \in \text{n-pred}(m))
\end{align*}
\]

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

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

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

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

- If $S$ is a set we denote $S^0$ the complete lattice obtained from $S$ by adjoining $\{\bot, \top\}$ to it, and imposing the ordering $\frac{1}{3} \leq x \leq \frac{2}{3}$ for all $x \in S$.

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

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

  - We assume that the meaning of an expression $e$ in the environment $e \in \text{Env}$ is given by $\text{val} | e \text{Expr} | e \text{Domain}$. The function $\text{val}$ in domain BExpr has the functionality:
    $$\text{val} | e \text{Expr} | e \text{Booll}.$$  

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

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

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

- The state transition function defines for each state a next state (we consider deterministic programs):
  $$\text{n-state} : \text{States} \to \text{States}$$

  - let $n$ be $\text{end}(\text{c}(s))$, $e$ be $\text{env}(s)$ within case $n$ in
    $$\text{Assignments}$$
    $$\text{Tests}$$
    $$\text{Junctions}$$
    $$\text{Exirs}$$

  (Each partial function $f$ on a set $S$ is extended to a continuous total function on the corresponding domain $S^5$ by $f(\bot) = \bot$, $f(\top) = \top$ and $f(x) = \bot$ if the partial function is undefined at $x$).

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

- Let $I$-states be the subset of initial states:
  $$I\text{-states} = \{\text{a} | \text{c}(s), \text{Env} | m \in \text{Entries}\}$$

Example:

```
(\text{x} = 1)
\text{if} x \leq 100 \text{ then true else false}
\text{if} x \leq x + 1
```
A "computation sequence" with initial state \( i_0 \in I\text{-states} \) is the sequence:
\[
  s_n = n\text{-state}^n(i_0) \quad \text{for } n = 0, 1, \ldots
\]
where \( \sigma^n \) is the identity function and \( \sigma^n f = f \circ \sigma^n \).

The initial to final state transition function:
\[
  n\text{-state}^\infty : \text{States} \to \text{States}
\]
is the minimal fixpoint of the functional:
\[
  \lambda F. (n\text{-state} \circ F)
\]
Therefore
\[
  n\text{-state}^\infty = Y_{\text{States}} \circ \lambda F. (n\text{-state} \circ F)
\]
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 \( Cq \) 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 \( P \):
\[
  Cq \subseteq \text{Contexts} = 2^{\text{Env}}
\]
\[
  Cq = \{ e | (\exists n \geq 0, i_n \in I\text{-states} | \langle q, e \rangle = n\text{-state}^n(i_n) ) \}
\]
The context vector \( Cv \) associates a context to each of the program points of a program:
\[
  Cv \subseteq \text{Context-Vectors} = \text{Arcs}^0 \to \text{Contexts}
\]
\[
  Cv = \lambda q. (e | (\exists i_n \in I\text{-states} | \langle q, e \rangle = n\text{-state}^n(i_n) )}
\]
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 \( r \),
\[
  \text{end}(q) = \text{origin}(r), \quad \begin{array}{c}
  \text{env-on}(r) \cap \text{env-on}(q) = \emptyset
  \end{array}
\]
From the definition of the state transition function we can prove the equation:
\[
  Cv(r) = n\text{-context}(r, Cv)
\]
where
\[
  n\text{-context} : \text{Arcs}^0 \times \text{Context-Vectors} \to \text{Contexts}
\]
is defined by:
\[
  n\text{-context}(r, Cv) = \begin{array}{l}
  \text{case } \text{origin}(r) \text{ in Entries} \\
  \begin{array}{l}
  \text{Assignments } \cup \text{ Tests } \cup \text{ Junctions } \\
  \bigcup e \in \text{env-on}(r) (n\text{-state}(\langle q, e \rangle ))
  \end{array}
  \end{array}
\]
\[
(\text{We define env-on : Arcs}^0 \to \text{States} = 2^{\text{Env}} \text{ to be (as, cond(r = x)(e)(env(s), \emptyset))} )
\]

Since the equation \( Cw(r) = n\text{-context}(r, Cv) \) must be valid for each arc, \( Cv \) is a solution to the system of "forward" equations:
\[
  Cv = F\text{-cont}(Cv)
\]
where
\[
  F\text{-cont} : \text{Context-Vectors} \to \text{Context-Vectors}
\]
is defined by:
\[
  F\text{-cont}(Cv) = \lambda r. n\text{-context}(r, Cv)
\]

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

\( F\text{-cont} \) is order preserving for the ordering \( \preceq \) of Context-Vectors which is defined by:
\[
  (Cv_1 \preceq Cv_2) \iff (\forall r \in \text{Arcs, } Cv_1(r) \preceq Cv_2(r))
\]
Hence it is known that \( F\text{-cont} \) has fixpoints, Tarski[55]. However, it is trivial to exhibit examples which show that these fixpoints are not always unique. Fortunately, it can be shown that \( Cv \) is included in any solution \( \mathcal{S} \) to the system of equations \( X = F\text{-cont}(X) \), \( (Cv \preceq \mathcal{S}) \).

Tarski[55] shows that this property uniquely determines \( Cv \) as the least fixpoint of \( F\text{-cont} \). Thus \( Cv \) can be equivalently defined by:

\[
  D1 : Cv = \lambda q. (e | (\exists n \geq 0, i_n \in I\text{-states} | \langle q, e \rangle = n\text{-state}^n(i_n) )
\]
or
\[
  D2 : Cv = Y_{\text{Context-Vectors}}(F\text{-cont})
\]
The concrete context vector \( Cv \) is such that for any program point \( q \in \text{Arcs of the program P} \),
\[
  (q) \text{ contains at least the environments } e \text{ which may be associated to } q \text{ during any execution of } P:
\]
\[
  (\exists i_n \in I\text{-states} | \langle q, e \rangle = n\text{-state}^n(i_n) ) \quad (e \in Cv(q))
\]
\[(q) \text{ contains only the environments } e \text{ which may be associated to } q \text{ during an execution of } P:
\]
\[
  (e \in Cv(q)) \quad (\exists i_n \in I\text{-states} | \langle q, e \rangle = n\text{-state}^n(i_n) )
\]

\( Cv \) 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 (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
\[
  I = (A\text{-Cont}, \preceq, \leq, \tau, \iota, \text{Int})
\]
where the set of abstract contexts is a complete \( o \)-semilattice with ordering \( \preceq \), \( (x \preceq y) \iff (x \circ y) \preceq (x \circ y') \).

This implies that \( A\text{-Cont} \) has a supremum \( \tau \). We suppose also \( A\text{-Cont} \) to have an infimum \( \iota \).
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 = \{Ar^s\} \rightarrow A-Cont.

Whatever (Cv, Cv') \in A-Cont may be, we define:
\[
Cv \rightarrow Cv' = \lambda \tau \cdot Cv'(r) \circ Cv''(r)
\]
\[
\triangleright \tau = \lambda r \cdot T \quad \triangleright \tau = \lambda r \cdot T
\]

\langle A-Cont, \leq, \tau, \triangleright \tau, \triangleright \tau \rangle can be shown to be a complete lattice. The function:
\[
\text{Int} : \{Ar^s\} \times A-Cont \rightarrow A-Cont
\]
defines the interpretation of basic instructions. If \{Cq\} \| 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 Int(r, C).

Int is supposed to be order-preserving:
\[
\forall a \in Ar^s, (Cv, Cv') \in A-Cont:
\]
\[
\{Cv \leq Cv'\} \rightarrow (\text{Int}(a, Cv') \leq \text{Int}(a, Cv))
\]

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

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

5.2 Typology of Abstract Interpretations

The restriction that "A-Cont" must be a complete semi-lattice is not drastic, since MacKeillé[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 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 to define completeness which ensures Int to have extreme fixpoints (see 8.4).

The result of an abstract interpretation was defined as a solution to forward (-) 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 (-) equations: a context may be related to its successors. Both systems (-, -) may also be combined.

Finally we usually consider a maximal (\tau) or minimal (\tau) solution to the system of equations, (by agreement, maximal and minimal are related to the ordering \leq defined by (x \leq y) \iff (x \land y = x)). However known examples such as Manna and Shamiyeh[73] show that the suitable solution may be somewhere between the extreme ones.

These choices give rise to the following types of abstract interpretations:

Examples:

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

5.3 Examples

5.3.1 Static Semantics of Programs

The static semantics of programs we defined in section 4 is an abstract interpretation:
\[
\text{gs} = \langle\text{Contexts}, u, \leq, \text{Env}, c, n{-\text{context}}\rangle
\]
where Contexts, u, \leq, Env, c, n{-context}, Context-Vectors, u, \leq, E{-context} respectively correspond to A-Cont, \leq, \tau, \triangleright \tau, \triangleright \tau, \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 \tau, if whenever computation reaches \tau, 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 be the set of expressions occurring in a program P. Abstract context will be sets of available expressions, represented by boolean vectors:
\[
\text{B}{\text{-vect}} : \text{Expr} \rightarrow \{\text{true}, \text{false}\}
\]

B{-vect} is clearly a complete boolean lattice. The interpretation of basic nodes is defined by:
\[
\text{avail}(r, Bv) = \begin{cases} 
\text{true}, & \text{if } r \in \text{origin}(r) \\
\text{false}, & \text{otherwise}
\end{cases}
\]

\[
\text{case } n \text{ in } 
\begin{align*}
\text{Entries } & \rightarrow \text{false} \\
\text{Assignments } \cup \text{Tests } \cup \text{Junctions } & \rightarrow \text{false} \\
\lambda e \cdot \text{(generated}(n)e) & \rightarrow \text{false} \text{ and By}(p)(e) \\
\text{p, n-pred}(n) & \rightarrow \text{false} \text{ and transparent}(n)(e))
\end{align*}
\]

(Noting is available on entry arcs. An expression e is available on arc \tau (exit of node n) if either the expression e is generated by n or for all predecessors of n, e is available on p and n does not modify arguments of e).

The available expressions are determined by the maximal solution (for ordering \text{false} \leq \text{false}) of the system of equations:
\[
\text{By} = \text{avail}(\text{By})
\]

241
Instead of the global hypothesis 6.0 we will use the following local hypothesis on the concrete and abstract interpretations of primitive language constructs:

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

6.5 and

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

These two hypotheses are in fact equivalent (lemma 12 in appendix J3). The following where.

5.3.3 Remarks

Our formal definition of abstract interpretations...
where \( n \text{-pred} \) defines Floyd[67]'s strongest post condition:

\[
\text{n-pred}(r, P_v) = \\
\text{let}(s \text{ be origin}(r), (p \text{ be a-pred(origin}(r))) \text{ within case } n \text{ in} \\
\text{Entries} \\ 
\implies \text{or} \\
\quad (P_v(q)) \\
\text{case } p \text{ in} \\
\quad \text{not test}(n) \\
\quad \text{case } a \text{-succ}(n) \implies \text{Pv}(p) \text{ and} \\
\quad \text{test}(n) \\
\quad \text{case } a \text{-succ}(n) \implies \text{Pv}(p) \text{ and} \\
\quad \text{not test}(n) \\
\text{esac} \\
\text{Assignments} \\ 
\text{let}(P \text{ be Pv}(p), (x \text{ be id}(n)), \\
\quad (e \text{ be expr}(n)) \text{ within} \\
\quad (\exists v \text{ Values} | P[v/x] \text{ and } x = e[v/x]) \\
\text{esac}
\]

The "invariants" of the program are defined by the least fixpoint of \( n \text{-pred} \) (least for ordering \( \sqsubseteq \) (\( \sqsubseteq \)), so that an invariant implies any other correct assertion). The deductive semantics is easily validated by proving that \( I_{DS} \sqsubseteq (\lambda_e \gamma) I_{DS} \) where:

\[
\alpha : \text{Contexts} \to \text{Pred} \\
\quad = \lambda C. \{ \text{for } t \text{ and } (x = e(x)) \} \\
\quad \in C \times \text{Ident} \\
\gamma : \text{Pred} \to \text{Contexts} \\
\quad = \lambda P. \{ e | P[e(x)/x], x \in \text{Ident} \}
\]

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

\[
\forall a \in \text{Arrows}, \forall P_v \in \text{Pred}, \\
(\alpha(a-context(a, \gamma(P_v))) \implies \text{n-pred}(a, P_v))
\]

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

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

\[
\forall q \in \text{Arrows}, \forall P_v \text{ the result of } I_{DS} \\
\{ (n \geq 0, \exists i_n \in I\text{-states} | <q,e> = n\text{-state}_i(n) ) \implies \text{Pv}(q) \implies \alpha(e) \}
\]

7. The Lattice of Abstract Interpretations

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

- reflexive: \((I \leq (I, I')) \) where \( i = \lambda x.x \) is the identity function,
- transitive: \((I \leq (\alpha_1, \gamma_1) I') \) and \((I' \leq (\alpha_2, \gamma_2), I')) \) imply \( I \leq (\alpha_1 \circ \alpha_2, \gamma_2 \circ \gamma_1) I' \).

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

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

is an equivalence relation. We have:

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

The proof gives some insight in the abstraction process:

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

2 - reciprocally,

If \( I \equiv (\alpha_1, \gamma_1) I' \) and \( (\alpha_2, \gamma_2) I' \) are the equivalence relation defined on \( I \) (properly speaking, on the set of abstract contexts of \( I \) by:

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

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

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

\[
\forall x' \in I', \exists x'' \in \gamma_1 \text{ such that } x' = \gamma_1(x'') \text{ and } x'' \in I
\]

Therefore \( \forall x' \in I' \), \( \exists x'' \in \gamma_1 \text{ such that } x'' \in I \)

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

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

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 isotope, its inverse \( \alpha_1^{-1} \) is isotope and \( \alpha_1 \text{(Int}(a, X) \text{)} \) is also a bijection between \( I \) and \( I' \)).

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

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

Example:

Let \( \text{P} \) 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 \( \alpha(S) = [\min(S), \max(S)] \). When \( S \) is infinite the bounds will eventually be \( -\infty = \gamma([a, b]) = \{ x | x \leq a \leq b \} \).

The abstract contexts are then, (Cousot[76]):
A further abstraction may be:
\[ \alpha((a, b)) = \begin{cases} \text{if } a + b \geq 0 \text{ then } a \text{ else } 0 & \text{if } a + b < 0 \end{cases} \]
\[\gamma(n) = [n, +\infty), \gamma(-) = [-\infty, 0), \gamma(\pm) = [-\infty, +\infty].\]

The abstract contexts are then:

\[ \Gamma_{CS} = \cdots -4 -3 -2 -1 0 1 2 3 4 \cdots \]

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

\[ \Gamma_{IP} = \cdots -3 -2 -1 0 1 2 3 \cdots \]
\[ \Gamma_{IR} = \cdots -3 -2 -1 0 1 2 3 \cdots \]
\[ \Gamma_{IR} \text{ is used by Kildall[75] for constant propagation.} \]
\[ \Gamma_{IR} \text{ might be used to apply the rules of signs. Both} \]
\[ \text{interpretations may be abstracted by:} \]

\[ \Gamma_{I} = \{\{a, b\}, \lambda(x, y), I, t, I, I, \lambda(a, C), I\} \]

where \( t \) is the relation which is always true. We have exhibited a sublattice of \( I \) which is:

8. Abstract Evaluation of Programs

The system of equations:
\[ CV : \text{Int}(CV) \]
resulting from an interpretation \( I = \langle A\text{-Cont}, \leq, \\cdot, t, \top, \text{Int} \rangle \) 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 (I.4 of Appendix 12):
\[ CV := (C := I; \text{until } C \text{ = Int}(C) \text{ do } C := \text{Int}(C) \text{ repeat} C) \]

8.1 Correctness

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

8.2 Termination

The abstract evaluation terminates if the Kleene's sequence is finite. This may be the case because A-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-Cont may be of finite length \( m \) (the length of any strictly increasing chain is bounded by \( m \), Kildall[73], Wegbreit[73]) or A-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 the Kleene's sequence is finite. Finally an infinite Kleene's sequence may be arbitrarily truncated (to get a lower bound of its limit), some induction principle (Sintzoff [75]) or heuristics (Katz and Manual[76]) may be used to pass to the limit, or approximate it, (Cousot[76]).

8.3 Efficiency

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

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

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

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

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

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

The main difficulty is to obtain 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 $\text{Kir}$ is clearly continuous (although it is not a complete morphism) since

$$\text{if } \text{Cv}_0 \preceq \text{Cv}_1 \preceq \ldots \preceq \text{Cv}_n \preceq \ldots \text{ then }$$

$$\max_{i=0}^{\infty} \Sigma_{p \in a\text{-pred}(n)} \text{Cv}_i(p) = \Sigma_{p \in a\text{-pred}(n)} \max_{i=0}^{\infty} \text{Cv}_i(p)$$

and

$$\max_{i \geq 0} (n_i \ast q) = (\max_{i \geq 0} (n_i)) \ast q.$$
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

\[ \text{As before, we define:} \]

9.1.3.3 \( A\text{-int} = \lambda q. A\text{-int}(q, CV) \)

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 \( S_0 = \hat{1}, \ldots, S_n \)

\[ \overset{\wedge}{A\text{-int}}(S_n), \ldots \]

We show that this sequence is increasing that is to say:

9.1.3.6 \( S_n \overset{\wedge}{A\text{-int}}(S_{n+1}), \forall n \geq 0 \)

Trivially for \( n = 0 \), \( S_0 = \hat{1} \overset{\wedge}{A\text{-int}}(S_0) \). For the
Let us note \([a, b]\) where \(a \leq b\) the predicate \(a \leq x \leq b\). The system of equations corresponding to the example is:

1. \(C0 = [1, 100]\)
2. \(C1 = [1, 1]\)
3. \(x := 1\)
4. \(C2 = C3 \cup C4\)
5. \(x \leq 100\)
6. \(C3 = [1, +\infty]\)
7. \(x := x + 1\)

* \(C3 = [1, 100]\)
* \(C4 = C3 \cup [1, 1]\)
* \([1, 100] \cup [1, 1]\)

\(\star \quad C4 = [2, 101]\)

Note: \(C1 \cup C4 = [1, 101] \leq C2 = [1, +\infty]\)
stop on that path.

\(C5 = C2 \cap [101, +\infty]\)
\(= [1, +\infty] \cap [101, +\infty]\)

\(\star \quad C5 = [101, +\infty]\)
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 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 a Upper
(However, we will not artificially truncate the decreasing sequence by imposing an arbitrary upper bound on $n$).

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

The limit of the descending sequence $S'_0 = \widetilde{t}, \ldots, S'_p = \overline{\text{Int}}P(\overline{t}), \ldots$ is an upper bound of the greatest fixpoint of $\text{Int}$.
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
Any of the AAS, TDS, DAS, TAS methods may yield a fixpoint $f_p$ which is not the fixpoint $lfp$ or $gfp$ of interest. None of these methods can improve $f_p$ 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-cont possesses a richer structure (i.e. for particular applications).

Furthermore, in the AAS, TDS, DAS, TAS sequences the term of rank $n$ is computed only as a function of the term of rank $n-1$, hence these are "separate steps" methods. One can as well imagine to use "bound steps" methods, where the term of rank $n$ is computed as a function of the terms of rank $n-1, n-2, \ldots, n-k$. 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-int and D-int could be more refined.

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.)


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


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


13. Appendix

We note <L, u, ≤, τ, I> a complete u-semilattice L, with partial ordering ≤, supremum τ and infimum I. These definitions are given in Birkhoff[61].

Note: L is a complete lattice.
(proof in Birkhoff[61], p. 49).

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

{∀(x, y) ∈ L², (x ≤ y) ⇒ {f(x) ≤ f(y)}}

⇒ {∀(x, y) ∈ L², {f(x) ≤ f(y)}}

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

(H2): Let ω be an order-preserving function from the complete semi-lattice <L, u, ≤, τ, I> in itself.

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

g = α(x) (x ∈ L) ∧ (x ∈ F(x))

(α, γ) is order preserving

(H2.1) γ : F → L

(H2.2) γ is order preserving

(H2.3) γ is order preserving

(H2.4) γ is order preserving

(H2.5) γ is order preserving

(H2.6) γ is order preserving

(Tarski[55], pp. 286–287). Note that the fixpoints of F need not form a sublattice of L.

We note g and ω the greatest and least fixpoints of F.

(H2): Let α be such that:

(H2.1) α : L → L

(H2.2) γ : F → L

(H2.3) γ is order preserving

(H2.4) γ is order preserving

(H2.5) γ is order preserving

(H2.6) γ is order preserving

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

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

Proof:

∀x ∈ L,

(γ(α(x))) = α(F(x)) by α = γ(x) in H3.1

(γ(F(x))) ≥ α(F(γ(x))) from H2.5

γ(F(x)) ≥ γ(α(γ(x))) from H2.6

γ is transitive

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

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

(L3): Let F : L → L be an order-preserving function from the semi-lattice <L, u, ≤, τ, I> in itself, and g and ω respectively the least and greatest fixpoints of F, then:

∀x ∈ L, {g ∨ F(x) ≥ x} ⇔ {g ≥ x}

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

∀x ∈ L, {x ∨ F(x) ≤ x} ⇔ {x ≤ x}

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

$$ (\alpha(g) \preceq g \text{ and } g \preceq \gamma(g) ) $$

Proof:

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

$$ g \preceq \alpha(g) \preceq \overline{\alpha(g)} \text{ trivially} $$

$$ g \preceq \overline{F(g)} \preceq \alpha(g) \text{ since } g = F(g) $$

$$ g \preceq \overline{F(\alpha(g))} \preceq \alpha(g) \text{ H3.1, } \preceq \text{ isotone, } \preceq \text{ transitive} $$

$$ g \preceq \alpha(g) \text{ L3} $$

$$ \gamma(g) \succeq \gamma(\alpha(g)) \text{ H2.4} $$

$$ \gamma(\overline{g}) \succeq g \text{ H2.6, } \preceq \text{ transitive.} $$

Q.E.D.

Replacing $<g, \preceq, \overline{g}, \succeq, \preceq, F, \overline{F}, \alpha, \gamma, H3.1, H2.4, H2.6>$ respectively by $<\overline{g}, \preceq, \alpha, \gamma, H3.1, H2.4, H2.6>$ in the above proof, we get the "dual" theorem:

(T2): $H_1, H_1, H_2, H_3$ imply that the least fixpoints $\ell$ and $\overline{\ell}$ of $F$ and $\overline{F}$ are related by:

$$ (\gamma(\overline{\ell}) \succeq \ell \text{ and } \ell \preceq \alpha(\ell)) $$

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 $f: D \to D$ is called continuous if whenever $X \subseteq L$ is directed, then $f(\bigcup \{x \mid x \in X\}) = \bigcup \{f(x) \mid x \in X\}$.

(H4): Let $F$ be a continuous function from the complete semi-lattice $<L, \preceq, \succeq, \top, \bot>$ in itself.

(H4'): Let $\overline{F}$ be a continuous function from the complete semi-lattice $<\overline{L}, \preceq, \succeq, \top, \bot>$ in itself.

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

(L4): $H4(\overline{H4})$ implies that $F$ ($\overline{F}$) has a least fixpoint $\ell(\overline{F})$ which is the limit $\bigcup_{i=0}^{\infty} F^i(\ell)$ of the Kleene's sequence $\ell \preceq F(\ell) \preceq \ldots \preceq F^n(\ell) \preceq \ldots$

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