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

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

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

A program denotes computations in some universe of objects. Abstract interpretation of programs consists in using that denotation to describe computations in another universe of abstract objects, so that the results of abstract execution give some informations on the actual computations. An intuitive example (which we borrow from Sintzoff [72]) is the rule of signs. The text \(-15 \times 17\) may be understood to denote computations on the abstract universe \((\ast), (-), (\#)\) where the semantics of arithmetic operators is defined by the sign of signs. The abstract program \(-15 \times 17\)

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.
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 \( a \leq b \) for all \( a, b \in S \).

- 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:
  \[
  \text{Env} = \text{Ident}^\bullet \to \text{Values}
  \]

We assume that the meaning of an expression \( \text{expr} \in \text{Expr} \) in the environment \( e \in \text{Env} \) is given by \( \text{val} \mid \text{Expr} \in \text{Env} \to \text{Values} \).

In particular the projection \( \text{val} \mid \text{Expr} \) of the function \( \text{val} \mid \text{domain} \text{Bexpr} \) has the functionality:

\[
\text{val} \mid \text{Bexpr} = \text{Bexpr} \to \text{Env} \to \text{Bool}.
\]

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

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

\[
\forall s \in \text{States}, s = \langle c(s), e(s) \rangle.
\]

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

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

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

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

\[
\text{n-state}(s) = \begin{cases}
  \text{let } n = \text{end}(c(s)), e = e(s) \text{ within case } n \in \\
  \text{Assignments} \Rightarrow \\
  \langle a\text{-succ}(n), e\text{val} \mid e\text{expr}(n) \mid (e) / (n) \rangle \Rightarrow \\
  \text{Tests} \Rightarrow \\
  \text{cond}(\text{val}[\text{test}(n)](e) \mid \text{Bexpr}, \\
  \langle a\text{-succ}(n), e, e\text{succ}(n), e \rangle) \Rightarrow \\
  \text{Junctions} \Rightarrow \\
  \text{Exits} \Rightarrow \text{exac}
  \end{cases}
\]

(Each partial function \( f \) on a set \( S \) is extended to a continuous total function on the corresponding domain \( S^\bullet \) by \( f(1) = 1, f(\top) = \top \) 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 s \in \text{Ident}^\bullet, \text{Env}(s) = \{\text{Values}\})

Let \( I\text{-states} \) be the subset of initial states:

\[
I\text{-states} = \{ \langle a\text{-succ}(n), \text{Env} \rangle | n \in \text{Entries} \}
\]
- A "computation sequence" with initial state $i_0 \in \text{I-states}$ is the sequence:
  $$s_n = \text{n-state}^n(i_0)$$
  for $n = 0, 1, \ldots$
  where $f^n$ is the identity function and
  $$f^{n+1} = f \circ f^n.$$

- The initial to final state transition function:
  $$\text{n-state}^\infty : \text{States} \rightarrow \text{States}$$

  is the minimal fixpoint of the functional:
  $$\lambda F . (\text{n-state} \circ F)$$

  Therefore
  $$\text{n-state}^\infty = \text{Y} \text{States} \rightarrow \text{States} (\lambda F . (\text{n-state} \circ F))$$
  where $\text{Y}_f(f)$ denotes the least fixpoint of $f : D \rightarrow D$, 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 $P$:

$$C_q \in \text{Contexts} = \mathcal{E}_\text{Env} \text{Env}$$

$$C_q = \{ e \mid (\exists n \geq 0, \exists i_n \in \text{I-states} \mid \langle q, e \rangle = \text{n-state}^n(i_n) \}$$

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

$$C_v = F_{\text{-cont}}(C_v)$$

where $F_{\text{-cont}} : \text{Context-Vectors} \rightarrow \text{Context-Vectors}$ is defined by:

$$F_{\text{-cont}}(C_v) = \lambda x . \text{n-context}(r, C_v)$$

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

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

$$\langle C_v_1 \leq C_v_2 \rangle \iff (\forall r \in \text{Arcs}, C_v_1(r) \leq C_v_2(r))$$

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 $C_v$ is included in any solution $\bar{C}$ to the system of equations $X = F_{\text{-cont}}(X)$, ($C_v \leq \bar{C}$). Tarski[55] shows that this property uniquely determines $C_v$ as the least fixpoint of $F_{\text{-cont}}$. Thus $C_v$ can be equivalently defined by:

$$D_1 : C_v = \lambda q . \{ e \mid (\exists n \geq 0, \exists i_n \in \text{I-states} \mid \langle q, e \rangle = \text{n-state}^n(i_n) \}$$

or

$$D_2 : C_v = \text{Y} \text{Context-Vectors}(F_{\text{-cont}})$$

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

(a) $C_v(q)$ contains at least the environments $e$ which may be associated to $q$ during any execution of $P$:
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 $= \text{Arcc}^0 \rightarrow A$-Cont.

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

$$Cv' \vee Cv'' = \lambda r . Cv'(r) \circ Cv''(r)$$

$$Cv' \sqcup Cv'' = \{ \forall r \in \text{Arcc}^0, Cv'(r) \leq Cv''(r) \}$$

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

$<A$-Cont, $\leq$, $\sim$, $\bot>$ can be shown to be a com-
6.5.5 Remarks

Our formal definition of abstract interpretations has the completeness property since the model ensures the existence of the particular solution to the system of equations and therefore defines at least some global property of the program. It must also have the consistency property, that is defined only correct properties of programs.

One can distinguish between syntactic and semantic abstract interpretations of a program. Syntactic interpretations are proved to be correct by reference to the program syntax (e.g. the algorithm for finding available expressions is justified by reasoning 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 \( \mathcal{I} = (A, \mathcal{C}, \mathcal{S}, \mathcal{T}, \mathcal{I}, \mathcal{I} \mathcal{C}) \) of a program is consistent with a "concrete" interpretation \( \mathcal{I} = (C, S, T, C, I) \) if the context vector \( CV \) resulting from \( \mathcal{I} \) is a correct approximation of the particular solution \( CV \) resulting from the more refined interpretation \( \mathcal{I} \). This may be rigorously defined by establishing a correspondence \( (\alpha : \text{abstraction}) \) between concrete and abstract context vectors, and inversely \( (\gamma : \text{concretization}) \), and requiring:

\[
\mathcal{I} \mathcal{C} \leq \gamma (\mathcal{C} \mathcal{V}) \quad \text{and} \quad \alpha (\mathcal{C} \mathcal{V}) \leq \mathcal{C} \mathcal{V}
\]

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

If \( f : D \to D' \) we note \( \beta = \mathcal{A}(\mathcal{C}) = D \) and \( \beta' = \mathcal{C}(\mathcal{C}) = D' \) and \( F : \beta \to \beta' = \lambda (x) f (\gamma (x)) \). We will suppose \( \alpha \) and \( \gamma \) to satisfy the following hypothesis:

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

\[
6.2 \quad \alpha \text{ and } \gamma \text{ are order-preserving}
\]

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

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

Intuitively, hypothesis 6.2 is necessary because context inclusion (that is property comparison) must be preserved by the abstraction or concretization 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 \( \gamma (\alpha (C)) \geq C \). Note that it is easy to prove properties corresponding to 6.1-6.4 for \( \alpha \) 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 constructs:

\[
\begin{align*}
\forall (a, x) \in \mathcal{A}(\mathcal{C}) \times \mathcal{A}(\mathcal{C}) \\
\gamma (\mathcal{C}(a, x)) \geq \mathcal{C}(a, \gamma (x))
\end{align*}
\]

6.5 and

\[
\begin{align*}
\forall (a, x) \in \mathcal{A}(\mathcal{C}) \times \mathcal{A}(\mathcal{C}) \\
\mathcal{C}(a, \alpha (x)) \geq \alpha (\mathcal{C}(a, x))
\end{align*}
\]

These two hypothesis are in fact equivalent (lemma 12 in appendix 11). The following schema illustrates 6.5, i.e. the idea of abstract simulation of concrete computations:

Suppose we want to compute the concrete output context \( C_0 \) (associated with arc \( a \)) resulting from concrete input contexts \( C_L \) : \( C_0 = \mathcal{C}(a, C_L) \). We can as well approximate this computation in the abstract universe, and get \( C_0' = \gamma (\mathcal{C}(a, \gamma (C_L))) \). 6.5 requires \( C_0' \) to contain at least \( C_0 \), that is \( C_0' \geq C_0 \). On the contrary we do not require \( C_0' \) to contain at most \( C_0 \), that is \( C_0' \leq C_0 \) is not compulsory.

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

Note that \( I \leq (\alpha, \gamma) \mathcal{I} \) imposes a local consistency of the interpretations \( I \) and \( \mathcal{I} \), at the level of primitive language constructs (6.5). Theorems \( T_1 \) and \( T_2 \) of Appendix 12 then prove 6.0 which defines the global consistency of \( I \) and \( \mathcal{I} \) at the program level.

In particular if we take

\[
I_{SS} = \mathcal{C}(\mathcal{C}, \mathcal{S}, \mathcal{E}, \mathcal{V}, \mathcal{O}, \mathcal{A}, \mathcal{C})
\]

any abstract interpretation \( \mathcal{I} \) of \( \mathcal{P} \), consistent with

\[
I_{SS} \leq (\alpha, \gamma) \mathcal{I}
\]

is consistent with the semantics of \( \mathcal{P} \), which implies:

\[
\forall a \in \mathcal{A}, \mathcal{C}(a, \gamma (x)) \text{ be the result of } \mathcal{I},
\]

\[
\{ \exists n, x \in \mathcal{S}, \text{true} = n-\text{scale} (i_n (x)) \}
\]

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 \( P(x_1, \ldots, x_n) \) \( \in \mathcal{P} \) over the program variables \( (x_1, \ldots, x_n) \in \mathcal{I} \) which are the free variables in the predicate. The abstract interpretation is then:

\[
I_{DS} = \{ \text{true, false, n-pred} \}
\]
where \( n\text{-pred} \) defines Floyd\cite{floyd67}'s strongest post condition:

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

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

\[
\begin{align*}
\text{Entries} & \Rightarrow (\forall x \in \text{Ident}, x = \text{i} \text{Values}) \\
\text{Junctions} & \Rightarrow \text{or} (P_v(q)) \\
\text{Tests} & \Rightarrow \text{case } r \text{ in } \\\n& \rightarrow (a\text{-succ}(t)(n) \Rightarrow P_v(p) \text{ and test}(n)) \\
& \rightarrow (a\text{-succ}(f)(n) \Rightarrow P_v(p) \text{ and not test}(n)) \\
\text{esac}
\end{align*}
\]

\[
\text{Assignments } \Rightarrow \]

\[
\text{let}(P \text{ be } P_v(p)), (x \text{ be } \text{id}(n)), (e \text{ be } \text{expr}(n)) \text{ within } \]

\[
(\lambda v \in \text{Values} | P_v(x) \text{ and } x = \text{e}(v/x))
\]

\[
\text{esac}
\]

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

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

\[
\alpha : \text{Contexts} \rightarrow \text{Pred} = \lambda C . (\text{or } (x \in C \text{ and } (x = e(x))))
\]

\[
\gamma : \text{Pred} \rightarrow \text{Contexts} = \lambda P . \{ e | P[e(x)/x], x \in \text{Ident} \}
\]

The main point is to justify Hoare\cite{hoare65}'s proof rules by showing:

\[
(\forall \alpha \in \text{Arcs}, \exists \gamma \in \text{SubPred}) \Rightarrow (\alpha(\text{context}(e, \gamma(P_v))) \Rightarrow n\text{-pred}(a, P_v))
\]

See Hoare and Lauer\cite{lauer74, liger75}. In particular Liger\cite{liger75} 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 \( I_{bg} \preceq (\alpha, \beta) I_{bg} \) has been proved, we know that the deductive semantics gives a valid proof technique, which will never permit a false theorem to be deduced:

\[
(\forall q \in \text{Arcs}, \exists P_v \text{ be the result of } I_{bg} \forall i_n \geq 0, \exists i_8 \in \text{I-states} | q, e = n\text{-state}(i_8)) \Rightarrow (P_v(q) \Rightarrow \alpha(e))
\]

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:

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

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:

\[
(\forall \alpha \in \alpha(\gamma(I))) \iff (\alpha(\gamma(I)))
\]

\( \forall x' \in I' \), each equivalence class \( \gamma_x \), \( \{ x \in I \mid x \in \gamma_x \} \) has a least upper bound which is \( \gamma_x(x') \). Hence the projection \( \gamma_x : \gamma_x(I') \) of \( \gamma_x \) on \( \gamma_x(I') \) is a bijection from the set \( \gamma_x(I') \) of representers of the equivalence classes on \( I \). Let us show now that under the hypothesis \( I \leq (\alpha, \gamma) I' \) and \( I' \leq (\alpha, \gamma) I' \), \( (\alpha, \gamma) \) is bijective:

\[
\alpha | \gamma(I') \text{ and } \gamma(I') \text{ are bijections, hence } \forall x' \in I', \text{ if } x \in I \Rightarrow \exists x'' \in \gamma(I) | x = (\alpha_1 \gamma(I)) (x'').
\]

Therefore, \( \forall x' \in I', \exists x'' \in \gamma(I) | x' = (\alpha_1 \gamma(I)) (x'') \). Thus

\[
(\alpha_1 | \gamma(I')) = (\alpha_2 | \gamma(I')) \text{ is a bijection between } \gamma(I) \text{ and } I'.
\]

Since \( (\alpha_2 | \gamma(I')) \) is a bijection between \( I \) and \( \gamma(I) \), the composition

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

\[
= (\alpha_1 | \gamma(I'))
\]

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

\[
= \text{Int}(a, \gamma(I)) \text{ Q.E.D.}
\]

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

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

This abstract evaluation terminates iff Kleene's sequence is finite. This may be the case because A-Cont is finite (e.g. type checking in ALGOL 68), 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 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 Maunel[76]) may be used to pass to the limit, or approximate it, (Cousot[76]).

8. Efficiency

In practice efficient versions of the Kleene's sequence are used. These consist of 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 $\omega$, with natural ordering $\leq$. The abstract interpretation:

$$L = \left< \mathbb{R}^+, \max, \leq, 0, \omega, A \right>$$

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

$$\text{Kir}(r, \mathcal{C}) = \text{let } n \text{ be origin(r) within case n in Entries \Rightarrow 1 \{unique \ entry \ node\} Junctions \cup Assignments \Rightarrow \bigwedge_{p \in A-\text{pred}(n)} \mathcal{C}(p) \text{ Tests} \Rightarrow \begin{cases} a-\text{suc}(n) \Rightarrow \mathcal{C}(a-\text{pred}(n)) \ast \text{Prob(test}(n) = \text{true}) \text{escac} \bigcap_{a-\text{suc}(n)} \mathcal{C}(a-\text{pred}(n)) \ast (1-\text{Prob(test}(n) = \text{true}) \text{escac} \end{cases}$$

The main difficulty is obtaining 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

$$\text{if } \mathcal{C}_0 \leq \mathcal{C}_1 \leq \ldots \leq \mathcal{C}_n \leq \ldots$$

then $\max_{i=0}^{\infty} \bigwedge_{a-\text{pred}(n)} \mathcal{C}_i(p) = \bigwedge_{a-\text{pred}(n)} \max_{i=0}^{\infty} \mathcal{C}_i(p)$ and $\max_{i=0}^{\infty} (n_{i+1} \ast q) = (\max_{i=0}^{\infty} n_i) \ast q$.

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 a limit of $0 + 1 + 1 + \ldots$

- Let $P$ be the program "begin while $T$ do $L$ end". The number $n$ of times the expression $\mathcal{I}$ is tested is given by the minimal solution to the equation $n = 1 + q \times n$ 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 $I = \langle A$-Cont, $\ast, \leq, 1, \tau, A \rangle$ be an interpretation of $P$. When the least fixpoint $\mathcal{C}$ of $\text{Int}$ is unreachable, we look for an upper bound $\mathcal{U}$ of $\mathcal{C}$, since according to the correctness requirement 6.0, $\mathcal{C} \leq \gamma(\mathcal{C})$ and $\mathcal{C} \leq \mathcal{U}$ implies $\mathcal{C} \leq \gamma(\mathcal{U})$.

9.1.1 Increasing Approximation Sequence

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

9.1.1.1 $(\forall n \geq 0, C = A$-Int$(0) \hat{\prec} \text{ and not}(\text{Int}(C) \preceq C) \rightarrow C \preceq (\text{Int}(C) \preceq A$-Int$(C))$.

9.1.1.2 Every infinite sequence $S_0, A$-Int$(S_0), \ldots, A$-Int$(S_n), \ldots$ is not strictly increasing.

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

9.1.1.3 $S_0 = 1$

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

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

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

Let $m$ be the least natural number (eventually infinite) such that $S_m = S_{m+1}$. We know from 9.1.1.3 that not$(\text{Int}(S_k) \preceq S_k)$, whence by definition of the ordering $\mathcal{E}$, $S_k \not\preceq \text{Int}(S_k) \preceq S_k$.

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

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

and therefore we conclude $S_{k+1} \preceq S_k$. We have $1, m \in \mathbb{N}$. Moreover 9.1.1.2 implies that $m$ is finite. Q.E.D.

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

$$\forall X \in A$-Cont, (\text{Int}(X) \preceq X) \rightarrow (\mathcal{C} \preceq X)$$

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

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

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:

$$\exists c \in \text{Arcs}, c \in \text{Arcs} \mid \text{Int}(q, c) \neq \text{Int}(q, c' c / r)$$

(e.g. in a forward system of equations the context associated to $q$ may only depend on the context associated with the immediate predecessor arcs of $q$). In the system of equations $\text{Cv} = \text{Int}(\text{Cv})$ we define a cycle to be a sequence $<q_1, \ldots, q_n>$ of arcs, such that $\forall i \in [1, n], \text{Cv}(q_{i+1})$ depends on $\text{Cv}(q_i)$ and $\text{Cv}(q_n)$ depends on $\text{Cv}(q_1)$. (e.g. in a forward interpretation a cycle corresponds to a loop in the program).

In any infinite strictly increasing Kleene's sequence $\text{Cv}_1, \ldots, \text{Cv}_n$, since $\text{Arcs}$ is finite there is some arc $q$ for which the sequence $\text{Cv}_q(q), \ldots, \text{Cv}_n(q)$, never stabilizes. Therefore $q$ must belong to a cycle or the contexts associated to $q$ transitively depend on the contexts associated to some other arc $r$ which itself belongs to a cycle. The sequence of contexts associated to any arc of that cycle never stabilizes. In order to avoid this phenomenon, we introduce:

- The binary operation $\lor$ called widening defined by:

9.1.3.1 $\lor : \text{A-Cont} \times \text{A-Cont} \rightarrow \text{A-Cont}$

9.1.3.2 $\lor(c, c') \in \text{A-Cont}^2, c \lor c' \in c \lor c'$

9.1.3.3 Every infinite sequence $s_0, \ldots, s_n$ of the form $s_0 = c_0, \ldots, s_n = s_{n+1} \in c_0, \ldots$, (where $c_0, \ldots, c_n$ are arbitrary abstract contexts) is not strictly increasing.

- The set $\text{W-arcs}$ of widening arcs, which is one of the minimal sets of arcs such that any cycle $<q_1, \ldots, q_n>$ of the system of equations $\text{Cv} = \text{Int}(\text{Cv})$ contains at least a widening arc: $\exists q \in [1, n] \mid q_i \in \text{W-arcs}$. (e.g. in a forward interpretation on a reducible program graph, W-arcs may be chosen to be the set of exit arcs of the junction nodes which are interval headers. On reducible graphs an arbitrary choice has to be made so that any loop of the program goes through a widening arc).

- The approximate interpretation $\text{A-Int} : \text{Arcs}^2 \times \text{A-Cont} \rightarrow \text{A-Cont}$ defined by:

9.1.3.4 $\text{A-Int} = \lambda (q, \text{Cv})$. If $q \in \text{W-arcs}$ then $\text{Cv}(q) \lor \text{Int}(q, \text{Cv})$ else $\text{Int}(q, \text{Cv})$.

As before, we define:

9.1.3.5 $\text{A-Int} = \text{A-Int} \cdot (\lambda q. \text{A-Int}(q, \text{Cv}))$

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

Let us consider a sequence $s_0 = \text{Cv}_1, \ldots, s_n = \text{A-Int}(s_n)$. We show that this sequence is increasing that is to say:

9.1.3.6 $s_n \leq \text{A-Int}(s_n), \forall n \geq 0$.

Trivially for $n = 0$, $s_0 = \text{Cv}_1 \leq \text{A-Int}(s_0)$. For the induction step, suppose the result to be true for $n \leq m$. Let us prove that:

$s_{m+1} = \text{A-Int}(s_{m+1})$.

If $q \in \text{W-arcs}$, then $\text{A-Int}(q, s_{m+1}) = \text{Int}(q, s_{m+1}) \lor \text{Int}(q, s_{m+1})$.

If $q \notin \text{W-arcs}$, then $\text{A-Int}(q, s_{m+1}) = \text{Int}(q, s_{m+1})$.

Finally $s_{m+1} \leq \text{A-Int}(s_{m+1})$.

An infinite sequence $s_0 = \text{Cv}_1, \ldots, s_n = \text{A-Int}(s_n)$ cannot be strictly increasing since otherwise there would exist some widening arc $q$ for which the sequence $s_q(q), \ldots, s_n(q)$ would never stabilize thus contradicting 9.1.3.3.

We now prove 9.1.1.7 that is to say that:

$\forall n \geq 0, s_n = \text{A-Int}(s_n)$

imply

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

If $q \in \text{W-arcs}$, we have $\text{A-Int}(q, s_n) = s_q(q) \lor \text{Int}(q, s_n)$. (see 9.1.3.5)

If $q \notin \text{W-arcs}$, then $\text{A-Int}(q, s_n) = \text{Int}(q, s_n) \lor \text{Int}(q, s_n)$ by 9.1.3.2. If now $q \notin \text{W-arcs}$ we must show:

$s_n(q) = \text{Int}(q, s_n)$.

which is true, from 9.1.3.6, Q.E.D.

9.2 Examples : Bounds of Integer Variables

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

246
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:

\[
\begin{align*}
(0) & \quad C_0 = \{1\} \\
(1) & \quad C_1 = \{1, 1\} \\
(2) & \quad C_2 = C_1 \cup C_4 \\
(3) & \quad C_3 = C_2 \cap [-\infty, 100] \\
(4) & \quad C_4 = C_3 + \{1, 1\} \\
(5) & \quad C_5 = C_2 \cap [101, +\infty] \\
\end{align*}
\]

Assignment statements are treated using interval arithmetic (e.g. \([i, j] + [k, l] = [i+k, j+l]\) naturally extended to include the case of the empty interval). Similarly, tests are treated using an "interval logic". Since there exist infinite Kleene's sequences (e.g. \(i, j \in [0, 0] < [0, 1] < ... < [0, +\infty]\) for the program \(x := 0\); while true do \(x := x+1\)), we must use an approximation sequence. Hence the results will be somewhat inaccurate but runtime subscript tests may be inserted in the absence of certainty.

Let us define the widening \(\triangledown\) of intervals by:

\[
\begin{align*}
-1 & \triangledown [i, j] \triangledown [k, l] = \begin{cases} 
\text{if } k < i & \text{then } -\infty \text{ else } i, \\
\text{if } k > j & \text{then } +\infty \text{ else } j 
\end{cases} \\
\end{align*}
\]

\(\triangledown\) satisfies the requirements of 9.1.3. According to 9.1.3,4 the system of equations is modified by:

\[
(2) \quad C_2 = C_2 \triangledown (C_1 \cup C_4)
\]

The corresponding approximation sequence is:

\[
\begin{align*}
+ & \quad C_1 = \{1\} \\
* & \quad C_2 = C_2 \triangledown (C_1 \cup C_4) \\
= & \quad \{1\} \triangledown \{1, 1\} \\
= & \{1, 1\} \\
C_3 = & \quad C_2 \cap [-\infty, 100] \\
= & \{1, 1\} \cap [-\infty, 100] \\
= & \{1, 1\} \\
C_4 = & \quad C_4 \triangledown \{1, 1\} \\
= & \{1, 1\} \triangledown \{1, 1\} \\
= & \{1, 2\} \\
C_5 = & \quad C_2 \cap [101, +\infty] \\
= & \{1, 2\} \cap [101, +\infty] \\
= & \{1, 100\} \\
C_3 = & \quad C_3 \triangledown \{1, 100\} \\
= & \{1, 1\} \triangledown \{1, 100\} \\
= & \{1, 100\} \\
C_4 = & \quad C_4 \triangledown \{1, 1\} \\
= & \{1, 1\} \triangledown \{1, 1\} \\
= & \{1, 1\} \\
C_5 = & \quad C_5 \triangledown \{101, +\infty\} \\
= & \{1, 100\} \triangledown \{101, +\infty\} \\
C_3 = & \quad C_3 \triangledown \{1, 100\} \\
= & \{1, 1\} \triangledown \{1, 100\} \\
= & \{1, 100\} \\
C_4 = & \quad C_4 \triangledown \{1, 1\} \\
= & \{1, 1\} \triangledown \{1, 1\} \\
= & \{1, 1\} \\
C_5 = & \quad C_5 \triangledown \{101, +\infty\} \\
= & \{1, 100\} \triangledown \{101, +\infty\} \\
C_3 = & \quad C_3 \triangledown \{1, 100\} \\
= & \{1, 1\} \triangledown \{1, 100\} \\
= & \{1, 100\} \\
C_4 = & \quad C_4 \triangledown \{1, 1\} \\
= & \{1, 1\} \triangledown \{1, 1\} \\
= & \{1, 1\} \\
C_5 = & \quad C_5 \triangledown \{101, +\infty\} \\
= & \{1, 100\} \triangledown \{101, +\infty\} \\
\end{align*}
\]

The final context on each arc is marked by a 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.

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

The ascending approximation sequence leads to an upper bound \(S_0 = \text{Int}^n(S)\) of the least fixpoint \(C^\infty\) of \(\text{Int}\). Moreover \(\text{Int}^n(S) \triangleleft S\). Since \(\text{Int}\) is order preserving, this implies that:

\[
\begin{align*}
S_0 & \triangledown \text{Int}(S) \triangleleft \text{Int}(S) \triangleleft \text{Int}(S) \triangleleft \text{Int}(S) \triangleleft \text{Int}(C^\infty) \\
\end{align*}
\]

If \(S\) is not a fixpoint of \(\text{Int}\) and the above descending sequence is finite (e.g. the lattice \(A\)-Cont satisfies the descending chain condition) its limit is a better approximation of \(C^\infty\) than \(S\).

When the sequence is infinite or slowly converging, one can among other solutions approximate its limit.

8.3.1 Decreasing Approximation Sequence

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

\[
\begin{align*}
\text{Int}^n(S) & \triangledown \text{Int}(S) \triangleleft \text{Int}(D) \\
\end{align*}
\]

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

\[
\begin{align*}
\text{Int}^n(S) & \triangledown D \triangleleft \text{Int}(D) \\
\end{align*}
\]

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

8.3.2 Truncated Decreasing Sequence

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

\[
\begin{align*}
3n \geq 0 & \quad \text{Int}(S) \triangledown D \triangleleft \text{Int}^n(S) \\
\end{align*}
\]

247
(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 defined such that:

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

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

The truncated decreasing sequence \(S'_0, S'_1, \ldots, S'_n, \ldots\) is recursively defined by:

9.3.2.3 \( S'_0 = S \)

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

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 \(S'_p = S'_p\). Trivially from 9.1.1:

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

If \(p > 0\) then \(S'_p \neq \text{Int}(S'_p)\), therefore \(S'_p \geq \text{Int}(S'_p)\). Then applying 9.3.2.1 we have:

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

But 9.3.2.3 implies \(S'_1 \neq \text{D-int}(S'_1)\), hence:

\[
S'_1 > S'_0 \geq \text{Int}(S'_0) \geq \text{CV}
\]

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

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

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

\[
\text{Int}(S'_k) \geq \text{Int}(S'_{k+1}) \geq \text{Int}^2(S'_{k-1}) \geq \text{Int}(\text{CV})
\]

By transitivity \(S'_k \geq \text{Int}(S'_k)\) and since 9.3.2.3 implies \(S'_k \neq \text{Int}(S'_k)\), we have from 9.3.2.1:

\[
S'_k \geq \text{D-int}(S'_k) = S'_{k+1} \geq \text{Int}(S'_{k+1})
\]

Since 9.3.2.3 implies \(S'_k \neq \text{D-int}(S'_k)\), we have:

\[
S'_k \geq S'_{k+1} \geq \text{Int}(S'_k) \geq \text{CV}
\]

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 Rene's Descending Sequence

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

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

9.3.4 Narrowing in Truncated Decreasing Sequence

By analogy with 9.1.3 we define now the narrowing operation in order to build a possible construction of \(\text{D-int}\) by local modifications to \(\text{int}\):

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

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

9.3.4.3 Every infinite sequence \(S_0, S_1, \ldots, S_i, \ldots\) of the form \(S_0 = C_0, S_1 = S_0 \Delta C_1, \ldots, S_i = S_{i-1} \Delta C_i, \ldots\) for arbitrary abstract contexts \(C_0, C_1, \ldots, C_n, \ldots\) is not strictly decreasing.

The approximated interpretation

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

9.3.4.4 \(\text{D-int} = \lambda(q, \text{CV}). \begin{cases} \text{Int}(q) & \text{if } q \notin \text{W-arcs} \\
\text{D-int}(q, \text{CV}) & \text{else}
\end{cases}\)

This definition of \(\text{D-int}\) trivially satisfies the requirement 9.3.2.1 since \(\text{CV} \geq \text{Int}(\text{CV})\) implies \(\text{CV}(q) \geq \text{Int}(q, \text{CV})\) for \(q \notin \text{W-arcs}\). However, if \(q \in \text{W-arcs} \) then 9.3.4.2 implies that \(\text{CV}(q) \geq \text{CV}(q) \Delta \text{Int}(q, \text{CV}) = \text{D-int}(q, \text{CV})\). Otherwise, if \(q \notin \text{W-arcs} \) then \(\text{CV}(q) \geq \text{Int}(q, \text{CV}) = \text{D-int}(q, \text{CV})\). Hence \(\text{CV} \geq \text{D-int}(\text{CV}) \geq \text{CV}\).

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

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

The ascending approximation sequence led to the approximate solution:

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

Let us define the narrowing \(\Delta\) of intervals by:

- \([i, j] \Delta [k, l] = \begin{cases} \text{null} & \text{if } i = -\infty \implies l = +\infty \\
\text{else min}(i, k, l) & \text{if } i = -\infty \text{ and } k < l \\
\text{else max}(i, k, l) & \text{if } i > l \text{ and } k < l
\end{cases}\)
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:

\[
\begin{align*}
C_2 &= C_2 \triangle (C_1 \cup C_4) \\
&= [1, +\infty) \triangle ([1, 1] \cup [2, 101]) \\
&= [1, +\infty) \triangle [1, 101]
\end{align*}
\]

* \( C_2 = [1, 101] \)

\[
\begin{align*}
C_3 &= C_2 \cap [-\infty, 100] \\
&= [1, 101] \cap [-\infty, 100] = [1, 100]
\end{align*}
\]
stop on that path.

\[
\begin{align*}
C_5 &= C_2 \cap [101, +\infty] \\
&= [1, 101] \cap [101, +\infty] = [101, 101]
\end{align*}
\]
exit.

On that example the approximate solution has been improved so that the least fixpoint is reached but this is not the case in general.

9.6 Dual Approximation Methods

The lattice \( \mathbb{A} \) may be partitioned as follows:

\( \mathbb{S} \) and \( \widehat{\text{Int}}(X) \)
non comparable

\( X \leq \widehat{\text{Int}}(X) < X \leq \widehat{\text{Int}}(X) \)

\( \bar{\mathbb{I}} \)

\( \mathbb{A} \)
\( \mathbb{S} \)
\( \mathbb{K} \)
\( \mathbb{G} \)
\( \mathbb{D} \)

\( \mathbb{S} \) and \( \mathbb{K} \) are the least and greatest fixpoints of \( \mathbb{S}_m \). The ascending (\( \mathbb{A} \)) and descending (\( \mathbb{D} \)) Kleene's sequences converge toward \( \mathbb{S} \) and \( \mathbb{G} \) respectively. These limits are reached when \( \mathbb{S}_m \) is continuous. When \( \mathbb{A} \) is infinite we have proposed to use an ascending approximation sequence (AAS) to approximate \( \mathbb{S} \). Its limit may be some fixpoint \( \mathbb{S} \), or some \( \mathbb{S}_m \) such that \( \mathbb{S}_m \leq \mathbb{S} \) and \( \mathbb{S}_m \geq \mathbb{S} \).

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 \( \mathbb{S}_n \) remain in the partition \( \{X | X \geq \mathbb{S}_n \} \), so that their limit \( \mathbb{G} \) is greater than \( \mathbb{S} \):

\[
\begin{align*}
\mathbb{A} \quad \mathbb{S}_n \quad \mathbb{G} \quad \mathbb{D} \quad \mathbb{K} \quad \mathbb{S} \quad \mathbb{G}
\end{align*}
\]

It is clear that the ascending approximation sequence AAS when starting from \( \mathbb{I} \) leads to an upper bound of the least fixpoint \( \mathbb{S} \) of \( \mathbb{S}_m \), and the truncated descending sequence TDS when starting from \( \mathbb{I} \) leads to an upper bound of the greatest fixpoint \( \mathbb{G} \). 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 \( \mathbb{S}_m \):

\[
\begin{align*}
\mathbb{A} \quad \mathbb{T} \quad \mathbb{S}_n \quad \mathbb{G} \quad \mathbb{T} \quad \mathbb{D} \quad \mathbb{T} \quad \mathbb{K} \quad \mathbb{S} \quad \mathbb{T} \quad \mathbb{G}
\end{align*}
\]
Any of the AAS, TDS, DAS, TAS methods may yield 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 "fixpoint improvement method" is necessary. It is our feeling that such a method could be designed only when considering that A-Dont possesses a richer structure (i.e. for particular applications).

Furthermore, in the AAS, TDS, DAS, TAS sequences the term of rank n is computed only as a function of the term of rank n-1, hence these are "separate steps" methods. One can as well imagine to use "bound steps" methods, where the term of rank n is computed as a function of the terms of rank n-1, n-2, ..., n-k. In this last case the Kleene's sequences may be used to compute the first k terms. After k steps more informations about the program would be available to heuristically accelerate the convergence so that the definition of A-int and D-int could be more refined.

Finally, going deeply into the comparism with numerical analysis methods, it is clear that some measure is necessary to control the accuracy of the result. Its definition would certainly also necessitate some additional properties of the abstract contexts.

10. Conclusion

It is our feeling that most program analysis techniques may be understood as abstract interpretations of programs. Let us point out global data flow analysis in optimizing compilers (Kildall[73], Horowitz[76]), Tompsett[76], Backhouse[75], Blum[67].

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.)


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).


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


13. Appendix

We note \(<L, \leq, s, t, \rhd>\) a complete \(\leq\)-semilattice \(L\), with partial ordering \(\leq\), supremum \(\rhd\) and infimum \(\rhd\). These definitions are given in Birkhoff[61].

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

We take \(f\) is isotonous, \(f\) is order-preserving or \(f\) is monotone to be synonymous, and mean:

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

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

\((H2)\): Let \(\bar{F}\) be an order-preserving function from the complete semi-lattice \(<L, \leq, \rhd, \rhd>\) in itself.

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

\[g = \alpha(x) \quad (x \in L) \wedge (x \leq F(x))\]
\[\xi = \alpha(x) \quad (x \in L) \wedge (F(x) \leq x)\]

(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 \(\xi\) the greatest and least fixpoints of \(F\).

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

\[(H2.1)\] \(\alpha : L \to L\)
\[(H2.2)\] \(\gamma : \bar{L} \to L\)
\[(H2.3)\] \(\alpha \) is order preserving
\[(H2.4)\] \(\gamma \) is order preserving
\[(H2.5)\] \(\forall x \in L, x \leq \alpha(y(x))\)
\[(H2.6)\] \(\forall x \in L, x \leq \gamma(y(x))\)

\((H3.1)\): \((H1)\), \((H2)\), and \(\forall x \in L, F(\alpha(x)) \geq \alpha(F(x))\)

\((H3.2)\): \((H1)\), \((H2)\), and \(\forall x \in L, \gamma(\alpha(x)) \geq \alpha(\gamma(x))\)

\((L1)\): \((H3.1)\) \iff \((H3.2)\)

Proof:
\(\forall x \in L, \gamma(\alpha(x)) \geq \alpha(\gamma(\alpha(x))) \implies (x = \gamma(x)) \) in \(H3.1\)
\(F(\gamma(x)) \geq \alpha(\gamma(\alpha(x))) \) from \(H2.5\)
\(\gamma(F(\gamma(x))) \geq \gamma(\alpha(F(\gamma(x)))) \) from \(H2.4\)
\(\gamma(F(\gamma(x))) \geq F(\gamma(x)) \) \(H2.6\) and transitivity.

\(\forall x \in L, \gamma(F(\alpha(x))) \geq F(\gamma(\alpha(x))) \implies (x = \gamma(x)) \) in \(H3.2\)
\(\gamma(\alpha(x)) \geq x \) \(H2.6\)
\(F(\gamma(\alpha(x))) \geq F(x) \) order preserving in \(H1)\).
\(\gamma(F(\alpha(x))) \geq F(x) \) transitivity
\(\alpha(\gamma(F(\alpha(x)))) \geq \alpha(F(x)) \) \(H2.3\)
\(F(\alpha(x)) \geq \alpha(F(x)) \) \(H2.5\)

Q.E.D.

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

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

\((L3)\): Let \(F : L \to L\) be an order-preserving function from the semi-lattice \(<L, \leq, \rhd, \rhd>\) in itself, \(k\) and \(g\) respectively the least and greatest fixpoints of \(F\), then:

\(\forall x \in L, \{g \cup F(x) \geq x\} \iff (g \geq x)\)

(The dual of this result is proved in Park[691], pp. 66). By duality:

\(\forall x \in L, \{k \cap F(x) \leq x\} \iff (k \leq x)\)
(T1): \( H_1, H_1, H_2, H_3 \) imply that the greatest fix-points \( g \) and \( \overline{g} \) of \( F \) and \( \overline{F} \) are related by:

\[
(\alpha(g) \preceq \overline{g}) \text{ and } (\overline{g} \preceq \gamma(\overline{g}))
\]

Proof:

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

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

Q.E.D.

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

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

\[
(\gamma(\overline{\xi}) \succeq \xi) \text{ and } (\xi \preceq \alpha(\xi))
\]

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 \( f(\bigcup \{x | x \in X\}) = \bigcup \{f(x) | x \in X\} \).

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

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

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

(L4) : H4(H4) implies that \( F (\overline{F}) \) has a least fix-point \( \overline{l}(\overline{F}) \) which is the limit \( \bigcup_{i=0}^{\infty} F^i(\bot) \) of the Kleene's sequence \( \bot \leq F(\bot) \leq \ldots \leq F^n(\bot) \leq \ldots \).

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

\[\]