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 Sintozoff [72]) is the rule of signs. The text $-1515 \times 17$ may be understood to denote computations on the abstract universe $\{(+), (-), (\#)\}$ where the semantics of arithmetic operations is defined by

\[ (+) + (+) = (+), \quad (+) \times (+) = (+), \quad (+) \times (-) = (-), \quad (-) \times (-) = (+), \quad (\#) \times (\#) = (\#), \quad (\#) + (\#) = (\#) \]

rule of signs. The abstract execution $-1515 \times 17 \Rightarrow -(+) \times (+) \Rightarrow (-) \times (+) \Rightarrow (-)$, proves that $-1515 \times 17$ 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. $-1515 + 17 \Rightarrow (-) + (+) \Rightarrow (-) \Rightarrow (\#)$). Despite its fundamentally incomplete results abstract interpretation allows the programmer or the compiler to answer questions which do not need full knowledge of program executions or which tolerate an imprecise answer, (e.g. partial correctness proofs of programs ignoring the termination problems, type checking, program optimizations which are not carried in the absence of certainty about their feasibility, ...).

Abstract program properties are modeled by a complete semilattice, Birkhoff[61]. Elementary program constructs are locally interpreted by order preserving functions which are used to associate a system of recursive equations with a program. The program global properties are then defined as one of the extreme fixpoints of that system, Tarski[55]. The abstraction process is defined in section 6. It is shown that the program properties obtained by an abstract interpretation of a program are consistent with those obtained by a more refined interpretation of that program. In particular, an abstract interpretation may be known to be consistent with the formal semantics of the language. Levels of abstraction are formalized by showing that consistent abstract interpretations form a lattice (section 7). Section 8 gives a constructive definition of abstract properties of programs based on constructive definitions of fixpoints. It shows that various classical algorithms such as Kildall [73], Wegbreit[75] compute program properties as limits of finite Kleene[52]'s sequences. Section 9 introduces finite fixpoint approximation methods to be used when Kleene's sequences are infinite, Cousot[76]. They are shown to be consistent with the abstraction process. Practical examples illustrate the various sections. The conclusion points out that abstract interpretation of programs is a unified approach to apparently unrelated program analysis techniques.

2. Summary

Section 3 describes the syntax and mathematical semantics of a simple flowchart language, Scott and Strachey[71]. This mathematical semantics is used in section 4 to build a more abstract model of the semantics of programs, in that it ignores the sequencing of control flow. This model is taken to be the most concrete of the abstract interpretations of programs. Section 5 gives the formal definition of the abstract interpretations of a program.

3. Syntax and Semantics of Programs

We will use finite flowcharts as a language independent representation of programs.

3.1 Syntax of a Program

A program is built from a set "Nodes". Each node has successor and predecessor nodes :

\[ n \text{-succ}, n \text{-pred} : \text{Nodes} \to \text{Nodes} \]

\[ (m \in n \text{-succ}(n)) \quad \Leftrightarrow \quad (n \in m \text{-pred}(m)) \]

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

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

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

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

** This work was supported by IRIA-SESORI under grants 75-035 and 75-160.
3.2 Semantics of Programs

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

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

$$Env = Ident^0 \rightarrow Values$$

- We assume that the meaning of an expression $expr \in Expr$ in the environment $e \in Env$ is given by $val \upharpoonright Expr\langle e \rangle$ so that:

$$val : Expr \rightarrow [Env \rightarrow Values].$$

In particular the projection $val \upharpoonright Bexpr$ of the function $val$ in domain $Bexpr$ has the functionality:

$$val \upharpoonright Bexpr : Bexpr \rightarrow [Env \rightarrow Bool].$$

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

$$\forall s \in States, s = \langle cs(s), env(s) \rangle.$$

- We use a continuous conditional function $cond(b, e_1, e_2)$ equal to $i \iff e_1$ or $e_2$ respectively as the value of $b$ is $i$, True, false or $\bot$. We also use $if b then e_1 else e_2 [\bot]$ to denote $cond(b, e_1, e_2)\upharpoonright \bot$.

- If $e \in Env, v \in Values, x \in Ident$ then $e \upharpoonright [v/x] = \lambda y.\;$ cond($y = x, v, e$)($y$).

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

$$n-state : States \rightarrow States$$

with

$$n-state(s) =$$

let $n$ be end$\upharpoonright cs(s)$, $e$ be $env(s)$ within $\forall case n in$

$$case n in$$

$$\begin{cases}
    Assignments & \rightarrow <a-succ(n), \upharpoonright \text{expr}(n) \upharpoonright e / id(n)> \\
    Tests & \rightarrow <\upharpoonright \text{test}(n) \upharpoonright e>, <a-succ\upharpoonright \text{test}(n), e> \\
    Juncions & \rightarrow <a-succ, n> \\
    Exits & \rightarrow s
\end{cases}$$

(Each partial function $f$ on a set $S$ is extended to a continuous total function on the corresponding domain $S^\uparrow$ by $f(1) = 1, f(T) = T$ and $f(x) = x$ if the partial function is undefined at $x$).

- Let $\bot_{Env}$ be the bottom function on $Env$ such that $\forall e : Ident^0, \bot_{Env}(x) = Values$.

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

$$I-states = \langle a-succ, \bot_{Env} \rangle \mid m \in Entries$$

Example:

```
\begin{tikzpicture}
  \node (x) at (0,0) {$x = 1$};
  \node (false) at (2,-2) {false};
  \node (true) at (2,2) {true};
  \node (a) at (2,1) {$x \leq 100$};
  \node (b) at (2,-1) {$x > 1$};

  \draw [->] (x) -- (false);
  \draw [->] (x) -- (true);
  \draw [->] (true) -- (a);
  \draw [->] (false) -- (b);
\end{tikzpicture}
```
A "computation sequence" with initial state $i_s \in I$-states is the sequence:

$$s_n = n\text{-state}^n(i_s)$$

for $n = 0, 1, ...$

where $\varepsilon^0$ is the identity function and $\varepsilon^{n+1} = f \circ \varepsilon^n$.

- 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

$$n\text{-state}^\infty = Y \text{States} \to \text{States} (\lambda F. (n\text{-state} \circ F))$$

where $Y(f)$ 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 sets 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 $P$:

$$C_q \in \text{Contexts} = \{ \varepsilon \mid (\exists n \geq 0, \exists i_s \in I\text{-states} \mid <q, \varepsilon> = n\text{-state}^n(i_s)\}$$

The context vector $C_v$ associates a context to each of the program points of a program $P$:

$$C_v \in \text{Context-Vectors} = \{ \lambda q. (\exists n \geq 0, \exists i_s \in I\text{-states} \mid <q, \varepsilon> = n\text{-state}^n(i_s))\}$$

According to the semantics of programs, the context $C_v(r)$ associated to arc $r$ is related to the contexts $C_v(q)$ at arc $q$ adjacent to $r$, $(\text{end}(q) = \text{origin}(r), \cdots)$. From the definition of the state transition function we can prove the equation:

$$C_v(r) = n\text{-context}(r, C_v)$$

where

$n\text{-context} : \text{Arcs}^2 \times \text{Context-Vectors} \to \text{Contexts}$

is defined by:

$$n\text{-context}(r, C_v) = \text{case } \text{origin}(r) \text{ in } \varepsilon \rightarrow$$

Since the equation $C_v(r) = n\text{-context}(r, C_v)$ 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

$$F\text{-cont} : \text{Context-Vectors} \to \text{Context-Vectors}$$

is defined by:

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

Context-Vectors is a complete lattice with union $\sqcup$ such that $C_v_1 \sqcup C_v_2 = n \cdot r. (C_v_1(r) \sqcup C_v_2(r))$. $F\text{-cont}$ is order preserving for the ordering $\preceq$ of Context-Vectors which is defined by:

$$\{C_v_1 \preceq C_v_2\} \iff (\forall r \in \text{Arcs}, C_v_1(r) \subseteq C_v_2(r))$$

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 $\sqcup$ to the system of equations $X = F\text{-cont}(X)$, $(C_v \subseteq \sqcup)$. 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:

$$D_1 : C_v = \lambda q. (\varepsilon \mid (\exists n \geq 0, \exists i_s \in I\text{-states} \mid <q, \varepsilon> = n\text{-state}^n(i_s)))$$

or

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

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 $\varepsilon$ which may be associated to $q$ during any execution of $P$:

$$\{i_s : 0, \exists i_s \in I\text{-states} \mid <q, \varepsilon> = n\text{-state}^n(i_s)\}$$

(b) $C_v(q)$ contains only the environments $\varepsilon$ which may be associated to $q$ during an execution of $P$:

$$\{\varepsilon \in C_v(q) \mid \forall i_s \geq 0, \exists i_s \in I\text{-states} \mid <q, \varepsilon> = n\text{-state}^n(i_s)\}$$

$C_v$ is merely a static summary of the possible executions of the program. However, our definitions $D_1$ or $D_2$ of $C_v$ cannot be utilized at compile time since the computation of $C_v$ consists in fact in running the program (for all possible input data). In practice compilers may consider states which 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.
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 $A-\text{Cont} = \text{Arcs}^0 \rightarrow A-\text{Cont}$.

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

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

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

\( \sim = \lambda r, T \) and \( \Downarrow = \lambda r, 1 \)

$$
\langle A-\text{Cont}, \sqsubseteq, \leq, \sim, \Downarrow, \sqsupset \rangle \text{ can be shown to be a complete lattice.}
$$

The function:

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

defines the interpretation of basic instructions. If \( \{ C(q) \mid q \in a-\text{prod}(n) \} \) is the set of input contexts of node \( n \), then the output context on exit arc \( r \) of \( n \) \( (r \in a-\text{succ}(n)) \) is equal to $\text{Int}(r, C)$.

\( \text{Int} \) is supposed to be order-preserving:

$$
\forall a \in \text{Arcs}, \forall (Cv', Cv'') \in A-\text{Cont}^2,

\{Cv' \sqsubseteq Cv'' \} \rightarrow \{ \text{Int}(a, Cv') \leq \text{Int}(a, Cv'') \}
$$

The local interpretation of elementary program constructs which is defined by $\text{Int}$ is used to associate a system of equations with the program. We define

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

It is easy to show that $\text{Int}$ is order-preserving. Hence it has fixpoints, \( \Upsilon \). Therefore the context vector resulting from the abstract interpretation \( I \) of program \( P \), which defines the global properties of the program, can be shown to be a solution of the system of equations.

\( \text{Int} = \text{Int}(Cv) \).

5.2 Typology of Abstract Interpretations

5.3 Examples

5.3.1 Static Semantics of Programs

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

$$
I_{\text{SS}} = \langle \text{Contexts}, \sqsubseteq, \text{Env}, \emptyset, n-\text{context} \rangle
$$

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

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
The determination of available expressions, back-dominators, intervals, ... requires a forward sys-

Instead of the global hypothesis 6.0 we will use the following local hypothesis on the concrete and
abstract interpretation of explicit lemmas and
where n-pred defines Floyd[67]'s strongest post condition:

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

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

case \( n \) in

\[ \text{Entries} \implies (\forall x \in \text{Ident, } x = i_{\text{Values}}) \]

\[ \text{Junctions} \implies \text{or} \]

\[ q\text{-pred}(n) \]

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

\[ (a\text{-succ}-t(n)) \implies Pv(p) \text{ and} \text{test}(n) \]

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

\[ \text{esac} \]

\[ \text{Assignments} \implies \]

\[ \text{let } (P \text{ be } Pv(p), (x \text{ be } \text{id}(n)), \]

\[ (e \text{ be } \text{expr}(n)) \text { within} \]

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

\[ \text{esac} \]

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

The deductive semantics is easily validated by pro-

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 \( I \) and \( I' \)

The proof gives some insight in the abstraction process:

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

\[ (I' \leq (\beta^{-1}, \beta)I)\} \]

2 - reciprocally,

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

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

\( \forall x' \in I' \), each equivalence class \( C_x = \{x \in I \mid \alpha_1(x) = x'\} \) has a least upper bound which is \( \gamma_1(x') \). Hence the projection \( \alpha \mid \gamma_1(I') \) of \( \alpha_1 \)

on \( \gamma_1(I') \) is a bijection from the set \( \gamma_1(I') \) of representatives of the equivalence classes on \( I \).

Let us show now that under the hypothesis

\( I \leq (\alpha_1, \gamma_1)I' \) and \( I' \leq (\alpha_2, \gamma_2)I' \), \( \alpha_1 \) is bi-
8. Abstract Evaluation of Programs

The system of equations:
\[ \text{Cv} : \text{Int}(\text{Cv}) \]
resulting from an interpretation \( \mathcal{I} = \langle A, \text{Cont}, \cdot, \leq, \tau, i, \text{Int} \rangle \) of a program \( P \) may be solved by "elimination" methods, (e.g. Tarjan[75]). Otherwise, one can use an "iterative" algorithm which computes Kleene's sequence (L4 of Appendix 12):
\[ \text{Cv} := (C := \mathcal{I}; \text{until } C = \text{Int}(C) \text{ do } C := \text{Int}(C) \text{ repeat}; C) \]

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 isotone function is continuous). Finally, if \( \text{Int} \) is only supposed to be
8.4 Example: Performance Analysis of Programs

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

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

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

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

\[
\text{Kir}(r, C_v) = \begin{cases} 
\text{let } n \text{ be origin}(r) \text{ within } & \\
\text{case } n \text{ in } & \\
\text{Entries} \Rightarrow 1 \{\text{unique entry node}\} \quad & \\
\text{Junctions} \cup \text{Assignments} \Rightarrow & \\
\text{Tests} \Rightarrow & \\
\text{case } r \text{ in } & \\
\{a-\text{succ}-r(n)\} \Rightarrow C_v(a-\text{pred}(n)) \times \frac{\text{Prob}(\text{test}(n) = \text{true})}{1 - \text{Prob}(\text{test}(n) = \text{true})} & \\
\text{esac} \quad & \\
\text{esac} & 
\end{cases}
\]

The main difficulty is to obtain the probability \( \text{Prob}(\text{test}(n) = \text{true}) \) 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

\[
\text{if } C_v_0 \leq C_v_1 \leq \ldots \leq C_v_n \leq \ldots
\]

then

\[
\max_{1 \leq i \leq n} p_c(a-\text{pred}(n)) = \sum_{1 \leq i \leq n} (\max(C_v_i(p)))
\]

and

\[
\max_{i \leq n} (n_i \times q) = (\max_{i \leq n} (n_i)) \times q.
\]

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., Temenbaum74). 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 \( I = (A-\text{Cont}, \preceq, \leq, 1, \tau, \text{Int}) \) be an interpretation of \( P \). When the least fixpoint \( C_v \) of \( \text{Int} \) is unreachable, we look for an upper bound \( U \) of \( C_v \), since according to the correctness requirement \( 0 \leq C_v \leq \gamma(C_v) \) and \( C_v \leq U \) implies \( C_v \leq \gamma(U) \).

9.1.1 Increasing Approximation Sequence

Let \( A-\text{Int} : A-\text{Cont} \to A-\text{Cont} \) be such that:

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

9.1.1.2 Every infinite sequence \( I, A-\text{Int}(I), \ldots, A-\text{Int}^n(I) \), \( \ldots \) is strictly increasing.

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

9.1.1.3 \( S_0 = I \)

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

We now prove that \( \exists m \) finite such that:

\[
S_0 \preceq S_1 \preceq \ldots \preceq S_m = S_{m+1} = \ldots
\]
9.1.2 Generalization of Kleene's Ascending Sequence

When A-Cont satisfies the ascending chain condition one can choose $\widetilde{\text{A-int}}$ to be $\text{Int}$ and therefore the approximation sequence generalizes Kleene's sequence and the related methods.

As before, we define:

$9.1.3.5 \quad \widetilde{\text{A-int}} = \lambda q . \text{A-int}(q, Cv)$

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

Let us consider a sequence $s_0$.
Let us note \([a, b]\) where \(a \leq b\) the predicate \(a \leq x \leq b\). The system of equations corresponding to the example is:

(0) \(C_0 = [1, 100]\)
(1) \(C_1 = [1, 1]\)
(2) \(C_2 = C_1 \cup C_4\)
(3) \(C_3 = C_2 \cap [-\infty, 100]\)
(4) \(C_4 = C_3 + [1, 1]\)
(5) \(C_5 = C_2 \cap [101, +\infty]\)

* \(C_3 = [1, 100]\)
* \(C_4 = C_3 + [1, 1]\)
* \([1, 100] + [1, 1]\)
* \(C_4 = [2, 101]\)
  Note: \(C_1 \cup C_4 = [1, 101] \leq C_2 = [1, +\infty]\)
  stop on that path.
  \(C_5 = C_2 \cap [101, +\infty]\)
  \(= [1, +\infty] \cap [101, +\infty]\)
  * \(C_5 = [101, +\infty]\)
  exit, stop.

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

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

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

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

The truncated decreasing sequence \( S_0', \ldots, S_n', \ldots \) is recursively defined by:

9.3.2.3 \( S_0' = S_m \)
\[ S_{n+1}' = \begin{cases} S_n' & \text{if } (S_n' \neq \text{int}(S_n')) \text{ and } (S_n' \neq \text{D-int}(S_n')) \\ \text{D-int}(S_n') & \text{else} \end{cases} \]

9.3.4 narrowing in truncated decreasing sequences

The limit of the descending sequence \( S_0' = \tilde{t}, \ldots, S'_n = \text{D-int}^n(\tilde{t}), \ldots \) is an upper bound of the greatest fixpoint of \( \text{int} \).

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

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

9.3.4.3 Every infinite sequence \( s_0, \ldots, s_n, \ldots \) of the form \( s_0 = C_0, s_1 = s_0 \Delta C_1, \ldots, s_n = s_{n-1} \Delta C_n, \ldots \) for arbitrary abstract contexts \( C_0, C_1, \ldots, C_n, \ldots \) is not strictly decreasing.

The approximated interpretation \( \text{D-int} : \text{Arcs}^3 \times \text{A-Cont} \to \text{A-Cont} \) is defined by:

...
Thus narrowing just discards infinite bounds and makes no improvement on finite bounds, it satisfies the requirements of 9.3.4. According to 9.3.4.4 the system of equations is modified by:

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

The descending approximation sequence is:

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

* \(C_3 = C_2 \cap [-\infty, 100]\)
* \(C_4 = [1, 101] \cap [-\infty, 100] = [1, 100]\) stop on that path.
* \(C_5 = C_2 \cap [101, +\infty]\)
* \(C_6 = [1, 101] \cap [101, +\infty] = [101, 101]\) exit.

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

9.5 Dual Approximation Methods

The lattice \(\widehat{X}\) may be partitioned as follows:

When \(X \geq Y\) we have noted \(X \rightarrow \rightarrow \rightarrow Y\). The truncated descending sequence TDS is fundamentally different from AAS, since it ensures that the successive approximations starting from \(S_m\) remain in the partition \(\{X | X \geq \widehat{\text{Int}}(X)\}\), so that their limit \(S_p\) is greater than \(\widehat{\text{fp}}\):
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 after consideration that AAS represents a shadow.

**Acknowledgements**

We wish to thank M. Sintzoff for stimulating discussions. We were very lucky to have F. Blanc do the typing for us.
(T1): $H_1$, $H_1$, $H_2$, $H_3$ imply that the greatest fixpoints $g$ and $\overline{g}$ of $F$ and $\overline{F}$ are related by:

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

**Proof:**

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

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

Q.E.D.

Replacing $<g, \overline{g}, \triangleright, \leq, \geq, F, \overline{F}, \sigma, \gamma, H3.1, H2.4, H2.6, \triangleright, H3.2, H2.3, H2.5>$ in the above proof, we get the "dual" theorem:

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

\[ \{ \gamma(\ell) \geq \ell \} \text{ and } \{ \ell \geq \alpha(\overline{\ell}) \} \]

According to Scott[71] a subset $X \subseteq L$ is called directed if every finite subset of $X$ has an upper bound (in the sense of $\leq$) belonging to $X$. (An obvious example of a directed subset is a non-empty ascending chain). A function $f: D \rightarrow D$ is called continuous if whenever $X \subseteq L$ is directed, then $f(\cup \{ x \mid x \in X \}) = \cup \{ f(x) \mid x \in X \}$.

(H4): Let $F$ be a continuous function from the complete semi-lattice $\langle L, \cup, \leq, \tau, 1 \rangle$ in itself.

(H4): Let $\overline{F}$ be a continuous function from the complete semi-lattice $\langle L, \overline{\cup}, \overline{\leq}, \overline{\tau}, 1 \rangle$ 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 $\cup \{ F^i(\ell) \}$ of the Kleene's sequence $1 \leq F(1) \leq \ldots \leq F^n(1) \leq \ldots$

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