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 -1515*17 may be understood to denote computations on the abstract universe \((+), (-), (\mathbb{Z})\) where the semantics of arithmetic operators is defined by the rule of signs. The abstract execution \(-1515*17 \Rightarrow (-1)\times(+) \Rightarrow (-)\times(+) \Rightarrow (-)\), 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. \(-1515+17 \Rightarrow (-1)\times(+1) \Rightarrow (-)\times(+) \Rightarrow (\mathbb{Z})\)). Despite its fundamentally incomplete results abstract interpretation allows the programmer or the compiler to answer questions which do not need full knowledge of program execution or which tolerate an imprecise answer, (e.g. partial correctness proofs of programs ignoring the termination problems, type checking, program optimizations which are not carried in the absence of certainty about their feasibility, ...).

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

2. Summary

Section 3 describes the syntax and mathematical semantics of a simple flowchart language, Scott and Strachey[71]. This mathematical semantics is used in section 4 to 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.

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:

\[\text{n-succ, n-pred : Nodes} \rightarrow \text{Nodes} \mid (m \in \text{n-succ}(n)) \Rightarrow (n \in \text{n-pred}(m))\]

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 Entries) has no predecessors and one successor, ((n-pred(n) = \emptyset) and \(|n\text{-succ}(n)| = 1\)).
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 $\{ot, \top\}$ 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 = $\{\text{true}, \text{false}\}$ and some other primitive domains.

- Environments are used to hold the bindings of identifiers to their values:
  $\text{Env} = \text{Ident}^0 \rightarrow \text{Values}$

  - We assume that the meaning of an expression $\text{expr}$ in the environment $e : \text{Env}$ is given by $\text{val(Expr)}(e)$ so that:
    $\text{val(Expr)}(e) = [\text{Env} \rightarrow \text{Values}]$.

  In particular the projection $\text{val} | \text{Expr}$ of the function $\text{val}$ in domain Bexpr has the functionality:
    $\text{val} | \text{Bexpr} : \text{Bexpr} \rightarrow [\text{Env} \rightarrow \text{Bool}]$.

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

  A state $(s, e : \text{States})$ consists in a control state $(\text{cs}(s))$ and an environment $(\text{env}(s))$, such that:
    $\forall s \in \text{States}, s = \langle \text{cs}(s), \text{env}(s) \rangle$.

- We use a continuous conditional function $\text{cond}(b, e_1, e_2)$ equal to $1$, $e_1$, $e_2$ or $\bot$ respectively as the value of $b$ is $1$, true, false or $\bot$. We also use if $b$ then $e_1$ else $e_2$ fi to denote $\text{cond}(b, e_1, e_2)$.

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

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

    $\text{n-state}(s) =$
    $\begin{cases}
      \text{let } n = \text{end}(\text{cs}(s)), e = \text{env}(s) \text{ within} \\
      \text{case } n \text{ in} \\
      \text{Assignments :} & <\text{a-succ}(n), \text{eval(Expr)(n)}(e) / \text{id}(n)> \\
      \text{Tests :} & <\text{a-succ}(n), e, \text{eval(Expr)(n)}(e) / \text{id}(n)> \\
      \text{Junctions :} & <\text{a-succ}(n), e> \\
      \text{Exits :} & s \\
    \end{cases}$

    (Each partial function $f$ on a set $S$ is extended to a continuous total function on the corresponding domain $S^\ast$ by $f(\bot) = 1$, $f(\top) = 0$ and $f(x) = 1$ if the partial function is undefined at $x$).

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

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

Example:

```
\begin{center}
\begin{tikzpicture}[node distance=2cm, on grid]
  \node (x) {$x = \text{true}$};
  \node (y) [below of=x] {$x \leq 100$};
  \node (z) [below of=y] {$x \geq x + 1$};
  \node (s) [above of=x] {$x = 1$};
  \node (f) [right of=s] {false};
  \node (t) [right of=y] {true};

  \draw[->] (x) -- (s);
  \draw[->] (x) -- (f);
  \draw[->] (y) -- (t);
  \draw[->] (y) -- (f);
  \draw[->] (z) -- (f);
  \draw[->] (z) -- (t);
\end{tikzpicture}
\end{center}
```
- A "computation sequence" with initial state \( s_0 \in \text{I-states} \) is the sequence:
  \[ s_n = N \text{-state}^n(s_0) \]
  for \( n = 0, 1, \ldots \)
  where \( N \) is the identity function and \( s_{n+1} = f \cdots f_n s_0 \).

- The initial to final state transition function:
  \[ n\text{-state}^\infty : \text{States} \rightarrow \text{States} \]
  is the minimal fixpoint of the functional:
  \[ \lambda F. (n\text{-state} \circ F) \]
  Therefore:
  \[ n\text{-state}^\infty = Y_{\text{States} \rightarrow \text{States}}(\lambda F. (n\text{-state} \circ F)) \]
  where \( Y(f) \) 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 \( P \):

\[
C_q \in \text{Contexts} = \{ e \mid \in \text{I-states} \}
\]

The context vector \( C_v \) associates a context to each of the program points of a program:

\[
C_v \in \text{Context-Vector} = \text{Arcs} \rightarrow \text{Contexts}
\]

According to the semantics of programs, the context \( C_v(r) \) associated to arc \( r \) is related to the context \( C_v(q) \) at arc \( q \) adjacent to \( r \),

\[
\text{end}(q) = \text{origin}(r), \quad \text{if} \quad q \rightarrow r.
\]

From the definition of the state transition function we can prove the equation:

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

where

\[
n\text{-context} : \text{Arcs} \times \text{Context-Vector} \rightarrow \text{Contexts}
\]

is defined by:

\[
\begin{align*}
\text{n-context}(r, C_v) &= \text{case} \text{origin}(r) \rightarrow \text{Entries} \rightarrow \{ \text{Env} \} \\
&\cup \text{env-on}(q)(n\text{-state}(q, e)) \\
&\cup \text{env-on}(q)(\text{pred}(\text{origin}(r)) \\
&\text{case}
\end{align*}
\]

We define \( \text{env-on} : \text{Arcs} \rightarrow \{ \text{States} \rightarrow 2^\text{Env} \} \) to be

\[
\lambda r. (\lambda s. \text{cond}(r = \text{eq}(s), (\text{env}(s), \emptyset)))
\]

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

\[
C_v = F\text{-cont}(C_v)
\]

where

\[
F\text{-cont} : \text{Context-Vector} \rightarrow \text{Context-Vector}
\]

is defined by:

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

Context-Vector is a complete lattice with identity \( \lambda r. (C_v(r)) \) such that \( C_v_1 \cup C_v_2 = \lambda r. (C_v_1(r) \cup C_v_2(r)) \).

\( F\text{-cont} \) is order preserving for the ordering \( \preceq \) of Context-Vector which is defined by:

\[
(C_v_1 \preceq C_v_2) \iff (\forall r \in \text{Arcs}, C_v_1(r) \preceq C_v_2(r))
\]

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 \( \exists C \) to the system of equations \( X = F\text{-cont}(X) \), \( (C_v \preceq C) \). 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:

\[
D_1 : C_v = \lambda q. (\lambda e. (\in \text{I-states} \rightarrow q.e \in n\text{-state}(i_s)))
\]

or

\[
D_2 : C_v = Y(F\text{-cont})(C_v)
\]

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 \( P \):

\[
(\exists i \in \text{I-states} \rightarrow q.e \in q.i_s)
\]

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

\[
(\exists e \in C_v(q) \rightarrow q.i_s)
\]

\( C_v \) 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 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

\[
I = \langle A\text{-Cont}, \leq, \epsilon, s, t, v, \square, \text{Int} \rangle
\]

where the set of abstract contexts is a complete \( o\)-semilattice with ordering \( \leq \), \( (s \leq y) \leftrightarrow ((x \leq y) \wedge (x \neq y)) \). This implies that \( A\text{-Cont} \) has a supremum \( \top \). We suppose also \( A\text{-Cont} \) to have an infimum \( i \).
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 = $\text{Arcs}^0 \rightarrow A$-Cont.

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

$Co' \oplus Co'' = \lambda r. Co'(r) \circ Co''(r)$

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

$\tilde{=} = \lambda r. 1$ and $\tilde{1} = \lambda r. 1$

$<A$-Cont, $\otimes$, $\oplus$, $\tilde{=}$, $\tilde{1}>$ can be shown to be a complete lattice. The function:

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

Examples:

1. $\text{Kil'An}'73$ uses $(\rightarrow, \sim)$; Moshreis'75 uses...
The determination of available expressions, back-
dominators, intervals, ... requires a forward sys-
tem of equations. Some global flow problems, notably
the live variables and very busy expressions
require propagating information backward through
the program graph, they are examples of backward
systems of equations.

6.5 Remarks

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

One can distinguish between syntactic and semantic
abstract interpretations of a program. Syntactic
interpretations are proved to be correct by refe-
cence to the program syntax (e.g., the algorithm for
finding available expressions is justified by rea-
soning on paths of the program graph). By contrast
semantic abstract interpretations must be proved
to be consistent with the formal semantics of the
language (e.g., constant propagation).

6. Consistent Abstract Interpretations

An "abstract" interpretation $I = \langle A$-Cont, $\approx, \preceq\rangle$
$\approx, \preceq, I, \text{Int}$ of a program is consistent with a "concrete"
interpretation $I = \langle C$-Cont, $\alpha, \beta, \preceq, I, \text{Int}$
if the context vector $CV$ resulting from $I$ is a cor-
rect approximation of the particular solver $CV$
resulting from the more refined interpretation $I$. This
may be rigorously defined by establishing a corres-
pondence ($\alpha : \text{abstraction}$) between concrete and ab-
stract context vectors, and inversely ($\gamma : \text{concreti-
}zation$), and requiring:

$$\langle \approx \rangle \in I(CV) \text{ and } \langle \gamma(CV) \rangle \preceq CV$$

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 $\overrightarrow{D} = \text{Arcs}^0 \to D$ and $\overrightarrow{D'} = \text{Arcs}^0 \to D'$
and if $f : \overrightarrow{D} \to \overrightarrow{D'} = \lambda \alpha.f(\alpha(D))$. We will suppose $\alpha$ and $\gamma$ to satisfy the following hypothesis:

6.1 $\alpha : C$-Cont $\to A$-Cont, $\gamma : A$-Cont $\to C$-Cont

6.2 $\alpha$ and $\gamma$ are order-preserving

6.3 $\forall x \in A$-Cont, $x \preceq \gamma(\gamma(x))$

6.4 $\forall x \in C$-Cont, $x \preceq \gamma(\gamma(x))$

Intuitively, hypothesis 6.2 is necessary because context inclusion (that is property comparison)
must be preserved by the abstraction or concreti-
ization process. 6.3 requires that concretization
introduces no loss of information. It implies that
$\alpha$ is surjective and $\gamma$ is injective. 6.4 introduces
the idea of approximation: the abstraction $\alpha(C)$ of
a concrete context $C$ may introduce some loss of
information so that when concretizing again $\gamma(\alpha(C))$
we may get a larger context $\gamma(\gamma(c)) \preceq C$. Note that
it is easy to prove properties corresponding to
6.1-6.4 for $\approx$ and $\beta$.

Instead of the global hypothesis 6.0 we will use the
following local hypothesis on the concrete and
abstract interpretations of primitive language con-
structs:

$$\forall (a, \overrightarrow{x}) \in \text{Arcs} \times A$-Cont,
\gamma(\text{Int}(a, \overrightarrow{x})) \preceq \gamma(\text{Int}(a, \overrightarrow{x}))$$

6.5 and

$$\forall (a, \overrightarrow{x}) \in \text{Arcs} \times C$-Cont,
\gamma(\text{Int}(a, \overrightarrow{x})) \preceq \gamma(\text{Int}(a, \overrightarrow{x}))$$

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

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

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

Note that $I \subset (\alpha, \gamma)\overrightarrow{I}$ imposes a local consistency
of the interpretations $I$ and $I$, at the level of pri-
mitive language constructs (6.5). Theorems 71 and
72 of Appendix 12 then prove 6.0 which defines the
global consistency of $I$ and $I$ at the program level.

In particular if we take

$I_{SS} = \langle \text{Contexts}, \preceq, \approx, \text{Env}, \emptyset, \text{a-context} \rangle$

any abstract interpretation $\overrightarrow{I}$ of $P$, consistent with

$I_{SS} (I_{SS} \subset (\alpha, \gamma)\overrightarrow{I})$ is consistent with the seman-
tics of $P$, which implies:

$$\forall q \in \text{Arcs}, \text{let } CV \text{ be the result of } \overrightarrow{I},
\forall n \in 0, \exists i_0 \in \text{states } \{q, e = n \text{-state}(i_0)\}
\Rightarrow (e \in CV(q))$$

As previously noticed, the abstract interpretations
will not in general be powerful enough to establish the
reciprocal.

Example: Deductive Semantics of Programs

Contexts will be predicates such as $P(x_1, \ldots, x_n)$
$\in \text{Pred}$ over the program variables $(x_1, \ldots, x_n) \in \text{Ident}$
which are the free variables in the predicate. The
abstract interpretation is then:

$I_{DS} = \langle \text{Pred}, \text{or}, \rightarrow, \text{true}, \text{false}, \text{n-pred} \rangle$
where \( n\text{-pred} \) defines Floyd's strongest post condition:

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

let \( s \) be \( \text{origin}(r) \), \( (p \models a\text{-pred}((\text{origin}(r)))) \)

\text{case } n \text{ in }

\begin{align*}
\text{Entries} & \implies (x \in \text{Ident}, x = i) \text{Values} \\
\text{Junctions} & \implies \text{or} (Pv(q)) \\
\text{Tests} & \implies \text{case } i \text{ in }
\begin{align*}
(a = \text{succ}-t(n)) \implies & \text{Pv}(p) \text{ and test}(n) \\
(a = \text{succ}+t(n)) \implies & \text{Pv}(p) \text{ and not test}(n)
\end{align*}
\text{esac}
\text{Assignments} \implies
\begin{align*}
\text{let } (P \text{ be } \text{Pv}(p)), (x \text{ be } \text{id}(n))\text{ within }
\begin{align*}
& (\forall v \text{ Values} \mid \text{Pv}(x) \text{ and } x = e[v/x]) \\
& (i \text{ be } \text{expr}(n))\text{ within }
\end{align*}
\\text{esac}
\end{align*}

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

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

\[ \alpha : \text{Contexts} \to \text{Pred} = \{c \mid \text{or } (x = e(x)) \}
\text{e }\in c, x \in \text{Ident} \]

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

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

\[ \{\forall a \in \text{ArCs}, \text{Pv} \in \text{Pre}, a(n\text{-context}(a, \gamma(Pv))) \to \text{n-pred}(a, \text{Pv}) \} \]

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

\[ \forall q \in \text{ArCs}, \text{let } \text{Pv} \text{ be the result of } I_{DG}^* \]

\[ \{n \geq 0, i \in I\text{-states} \mid q \in \text{n-state } n(i) \} \]

\[ \to (\text{Pv}(q) \to \text{a}(e)) \]

7. The Lattice of Abstract Interpretations

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

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

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

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

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

is an equivalence relation. We have:

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

The proof gives some insight in the abstraction process:

\[ 1 - \{I \models (\beta)I'\} \to \{I \preceq (\beta \circ \beta^{-1})I' \text{ and } I' \preceq (\beta^{-1} \circ \beta)I\} \]

2. Reciprocally,

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

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

\( x \models y \), each equivalence class \( \gamma_{\alpha}(x) \) of \( \gamma_{\alpha} \) on \( \gamma_{\alpha}(I') \) is a bijection from the set \( \gamma_{\alpha}(I') \) of representatives of the equivalence classes on \( I \).

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

\[ (\alpha_1 \circ \gamma_1)I' \text{ and } \alpha_2 \circ \gamma_2 I' \]

are bijections, hence \( x \models y \in I' \), \((\forall x \in I \equiv \gamma_{(\alpha_1)}(I')(x)) \). Likewise, \( x \equiv \gamma_{(\alpha_1)}(I') \)

\[ \to \gamma_{(\alpha_1 \circ \alpha_2)}(I')(x) = \gamma_{(\alpha_1 \circ \alpha_2)}I' \]

\[ \gamma_{(\alpha_1 \circ \alpha_2)}(I') \]

\[ \text{thence } \alpha_1 \circ \gamma_1 \text{ is a bijection between } \gamma_2(I') \text{ and } I' \to \gamma_2(I') \text{ is a bijection between } I \text{ and } I' \text{, the composition } \]

\[ (\alpha_1 \circ \gamma_1)(I') \]

\[ (\alpha_1 \circ \gamma_1)(I') \]

\[ = (\alpha_2 \circ \gamma_2)(I') \]

\[ \text{is a bijection between } I \text{ and } I' \text{, hence } \alpha_1 \text{ is a bijection between } I \text{ and } I' \text{ which is trivially an algebraic morphism. } (\alpha_1 \text{ is isotope, its inverse } \alpha_1^{-1} = \gamma_1 \text{ is isotope and } \alpha_1(\text{Int}(a, x)) \}

\[ = \text{Int}'(a, \gamma_1(x)) \text{ Q.E.D.} \]

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_{DG} as I is then a lattice, (with ordering \( \preceq \)) which is isomorphic with a subset of the lattice of equivalence relations on Contexts.

Example:

Let P be a program with a single integer variable, (the generalization is obvious). Environments will be integers (the value of the variable). Contexts are sets of integers (the set of values at some program point).

A context S may be abstracted by a closed interval \( \alpha(S) = [\min(S), \max(S)] \). When S is infinite the bounds will eventually be \( +\infty \) and \( -\infty \).

The abstract contexts are then, (Cousot[76]):

243
A further abstraction may be:

\( \alpha([a, b]) = \text{if } a \leq b \text{ then } a \text{ else if } a \geq 0 \text{ then } + \text{ else if } b \leq 0 \text{ then } - \text{ else } 0 \text{.} \)

\( \gamma(+) = [0, +\infty], \gamma(-) = [-\infty, 0[, \gamma(0) = [-\infty, +\infty]. \)

The abstract contexts are then:

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

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

\[ \Gamma_{I} = \Gamma_{RS} \]

8. Abstract Evaluation of Programs

The system of equations:

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

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

8.1 Correctness

If \( \text{Int} \) is supposed to be a complete morphism \((i.e., \text{infinitely distributive over } \cdot)\) then \( \text{Cv} \) is the least fixpoint of \( \text{Int} \) \((e.g., \text{Kildall}[75]), \) 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 \( \text{Int} \) \((e.g., \text{Wegbreit}[75]), \) since in a well-founded semi-lattice, any isotone function is continuous). Finally, if \( \text{Int} \) is only supposed to be isotone, \( \text{Cv} \) is an approximation \((?)\) of the least fixpoint \((e.g., \text{Kam and Ullman}[75])\).

8.2 Termination

The abstract evaluation terminates iff Kleene's sequence is finite. This may be the case because \( \text{A-Cont} \) is finite \((e.g., \text{type checking in ALGOL 60,})\).
8.4 Example: Performance Analysis of Programs

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

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

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

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

$$\text{Kir}(r, C_v) = \text{let } n \text{ be origin}(r) \text{ within case } n \text{ in Entries} \Rightarrow 1 \text{ (unique entry node)}$$

$$\text{Junctions} \cup \text{Assignments} \Rightarrow \bigoplus_{p \in a-pred(n)} C_v(p)$$

$$\text{Tests} \Rightarrow \begin{cases} \text{case } r \text{ in } (a-\text{succ-}r(n)) \Rightarrow C_v(a-\text{pred}(n)) \ast \frac{\text{Prob}(\text{test}(n) = \text{true})}{\text{Prob}(\text{test}(n) = \text{false})} \\ \text{else} \end{cases}$$

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 (hypothesis on the input data).

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

$$\text{if } C_v_0 \geq C_v_1 \geq \ldots \geq C_v_n \geq \ldots$$

$$\text{then } \max_{1 \leq i \leq n} \bigoplus_{p \in a-pred(n)} C_v_i(p) = \max_{1 \leq i \leq n} \bigoplus_{p \in a-pred(n)} C_v_i(p)$$

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 an upper limit of $0 + 1 + 2 + \ldots$
- Let $P$ be the program "begin while T do L end". The number $n$ of times the expression $T$ is tested is given by the minimal solution to the equation $n = 1 + q + n$ where $q$ is the probability of $T$ to be true. $n$ may be determined by the limit of Kleene's sequence:

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

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 Jacob'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 $\tilde{I}$ ($1 \leq \tilde{I}$) may be used for that purpose (e.g. Tennenbaum '74). It is often better to make approximations in $\tilde{I}$, 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 $I = \langle A-\text{Cont}, \ast, \leq, 1, \tau, \text{Int} \rangle$ be an interpretation of $P$. When the least fixpoint $C_v$ of $\text{Int}$ is unreachable, we look for an upper bound $U_B$ of $C_v$, according to the correctness requirement:

$$\text{if } C_v \not\leq \gamma(C_v) \text{ and } C_v \not\leq U_B \text{ implies } C_v \not\leq \gamma(U_B).$$

9.1.1 Increasing Approximation Sequence

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

$$9.1.1.1 \quad (\forall m \geq 0, C = \text{A-Int}^m(C) \text{ and not } (\text{Int}(C) \not\leq C) \Rightarrow (C \not\leq \text{Int}(C) \not\leq \text{A-Int}(C)).$$

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

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

$$9.1.1.3 \quad S_0 = 1$$

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

We now prove that $3n$ finite such that:

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

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 (Int(S_k) $\not\leq S_k$). Whence by definition of the ordering $\preceq$ if $S_k \not\preceq$ Int(S_k) $\not\preceq S_k$.

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

$$S_{k+1} = A-\text{Int}(S_k) \not\preceq \text{Int}(S_k) \not\preceq S_k$$

and therefore we conclude $S_{k+1} \preceq S_k$. $\forall k \in [0, m[,\ldots$. Moreover 9.1.1.2 implies that $m$ is finite. Q.E.D.

Let $C_v$ be the least fixpoint of $\text{Int}$, it is the greatest lower bound of the set of $X \in A-\text{Cont}$ such that $\text{Int}(X) \not\preceq X$ (Tarski '55) hence:

$$\forall X \in A-\text{Cont}, (\text{Int}(X) \not\preceq X) \Rightarrow (C_v \not\preceq X)$$

Since $S_m = S_{m+1}$ we have $\text{Int}(S_m) \not\preceq S_m$ and therefore $C_v \not\preceq S_m$. $S_m$ is a correct approximation of $C_v$. 245
9.1.2 Generalization of Kleene's Ascending Sequence

When A-Cont satisfies the ascending chain condition one can choose \( A\text{-Int} \) 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 \( A\text{-Int} \) in 9.1.1 is global. We now indicate a way to construct \( A\text{-Int} \) 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:

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

(e.g. in a forward system of equations the context associated to \( r \) may only depend on the context associated to \( r \).

As before, we define:

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

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

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

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

\[
S_{m+1} \preceq A\text{-int}(S_{m+1})
\]

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

If \( q \in \text{w-arcs} \), then \( A\text{-int}(q, S_{m+1}) - S_{m+1}(q) \not\in \text{Int}(q, S_{m+1}) \geq S_{m+1}(q) \preceq \text{Int}(q, S_{m+1}) \)

\[
S_{m+1}(q) \preceq A\text{-int}(q, S_{m+1})
\]

\[
\forall q \in \text{Arcs}.
\]
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. \(C_0 = \{ [], \} \)
2. \(C_1 = \{ [], \} \)
3. \(C_2 = \{ [], \} \)
4. \(C_3 = \{ [1, 100] \}
5. \(C_4 = \{ [1, 1] \}
6. \(C_5 = \{ [1, 100], [1, 1] \}

Assignment statements are treated using interval arithmetic (e.g., \([i, j] + [k, \ell] = [i + k, j + \ell]\)) 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., \([i, j] < \infty \leq [0, 1] < \cdots < [0, \infty] \) 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 \(\vee\) of intervals by:

\(-[i, j] \vee [k, \ell] = [\text{if } k < i \text{ then } -\infty \text{ else } i, \text{ if } \ell > j \text{ then } +\infty \text{ else } j] \)

\(\vee\) satisfies the requirements of 9.1.3. According to 9.1.3.4 the system of equations is modified by:

2. \(C_2 = C_2 \vee (C_1 \cup C_4)\)

The corresponding approximation sequence is:

* \(C_0 = \{ \} \)
* \(C_1 = \{ [1, 1] \}
* \(C_2 = \{ [1, 1], \} \)
* \(C_3 = \{ [1, 100] \}
* \(C_4 = \{ [1, 100], [1, 1] \}
* \(C_5 = \{ [1, 100], [1, 1], [1, 100], [1, 100] \}

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 Decreasing Approximation of the Least Fixpoint Starting from an Upper Bound

The ascending approximation sequence leads to an upper bound \(S_m = \text{Int}^\wedge (s)\) of the least fixpoint \(CV\) of \(\text{Int}\) or \(\text{Int}^\wedge\). Moreover \(\text{Int}^\wedge (S_m) \leq S_m\). Since \(\text{Int}\) is order preserving, this implies that:

\[ S_m \geq S_m \geq \cdots \geq S_m \geq CV \]

If \(S_m\) 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 \(CV\) than \(S_m\). When the sequence is infinite or slowly converging, one can among other solutions approximate its limit.

9.4 Decreasing Approximation Sequence

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

\[ \text{Int}^{\wedge n} (S_m) \geq \text{Int}^{\wedge n} (S_m) \geq CV \]

In order to accelerate the convergence, we should for the next step find an approximation \(D\) such that \(\text{Int}^{\wedge n+1} (S_m) \geq D \geq CV\). But not knowing \(CV\), this characterization is very weak since \(D\) could be chosen incorrectly that is to say less than \(CV\) or non comparable with \(CV\). The fact that \(CV\) is the greatest lower bound of the set of \(X \in A\)-Cont such that \(\text{Int}^{\wedge n} (X) \geq CV\) gives a correctness criterion for the choice of \(D\) when \(CV\) is unknown, we must have:

\[ \text{Int}^{\wedge n} (S_m) \geq D \geq \text{Int} (D) \]

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

9.3.8 Trimmed Decreasing Sequence

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

\[ \exists n \geq 0 \mid \text{Int} (S_m) \geq D \geq \text{Int}^{\wedge n} (S_m) \]

247
(However, we will not artificially truncate the decreasing sequence by imposing an arbitrary upper bound on n.)

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

9.3.2.1 \( \forall c \in A\text{-Cont} \)

\[ (c \geq \text{Int}(c)) \Rightarrow (c \leq D\text{-Int}(c) \leq \text{Int}(c)) \]

9.3.2.2 \( \forall c \in A\text{-Cont} \), every infinite sequence \( c_0, D\text{-Int}(c_0), \ldots, D\text{-Int}(c_n), \ldots \) is not strictly decreasing.

The truncated decreasing sequence \( S_0, S_1, \ldots, S_n, \ldots \) is recursively defined by:

9.3.2.3 \( S_0 = S_m \)

\[ S_{n+1} = \begin{cases} S_n & (S_n \neq \text{Int}(S_n)) \text{ and } (S_n \neq D\text{-Int}(S_n)) \\ \text{else} & S_n \end{cases} \]

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

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

\[ S_0 = S_m \geq \text{Int}(S_0) \geq CV \]

If \( p > 0 \) then \( S_p \neq \text{Int}(S_p) \), therefore \( S_0 \geq \text{Int}(S_0) \).

Then applying 9.3.2.1 we have:

\[ S_0 \geq D\text{-Int}(S_0) = S_1 \geq \text{Int}(S_0) \geq CV \]

But 9.3.2.3 implies \( S_0 \neq D\text{-Int}(S_0) \), hence:

\[ S_0 > S_1 \geq \text{Int}(S_0) \geq CV \]

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

\[ S_k < S_{k+1} \geq \text{Int}(S_k) \geq CV \]

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

\[ \text{Int}(S_{k+1}) \geq \text{Int}(S_k) \geq \text{Int}(S_{k+1}) \geq \text{Int}(CV) \]

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:

\[ S_k \geq D\text{-Int}(S_k) = S_{k+1} \geq \text{Int}(S_k) \]

Since 9.3.2.3 implies \( S_k \neq D\text{-Int}(S_k) \) we have:

\[ S_k > S_{k+1} \geq \text{Int}(S_k) \geq CV \]

By recurrence on \( k \) the result is true for \( k < p \).

Moreover 9.3.2.2 implies that \( p \) is finite. Q.E.D.

9.3.3 Generalization of Kleene's Descending Sequence

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

9.3.4 Narrowing in Truncated Decreasing Sequences

By analogy with 9.1.3 we define now the narrowing operation in order to build a possible construction of \( D\text{-Int} \) 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 \( \forall (c, c') \in A\text{-Cont}^2 \),

\[ (c \geq c') \Rightarrow (c \geq \Delta(c, c') \geq c') \]

9.3.4.3 Every infinite sequence \( s_0, s_1, s_2, \ldots \)

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

9.3.4.4 \( D\text{-Int} = \lambda(q, CV).\) if \( q \in W\text{-Arcs} \)

\[ CV(q) \geq \text{Int}(q, CV) \]

else

\[ q \]

and \( D\text{-Int} = \lambda CV. (\lambda q. D\text{-Int}(q, CV)) \)

This definition of \( D\text{-Int} \) trivially satisfies the requirement 9.3.2.1 since \( CV \geq \text{Int}(CV) \) implies \( CV(q) \geq \text{Int}(q, CV) \), \( q \in W\text{-Arcs} \).

If \( q \in W\text{-Arcs} \) then 9.3.4.2 implies that \( CV(q) \geq CV(q) \Delta \text{Int}(q, CV) = D\text{-Int}(q, CV) \).

Otherwise, if \( q \notin W\text{-Arcs} \) then \( CV(q) \geq \text{Int}(q, CV) = D\text{-Int}(q, CV) \).

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:

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

The ascending approximation sequence led to the approximate solution:

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

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

\[
\begin{align*}
&[-, -] \Delta [k, k] = \emptyset \\
&[i, j] \Delta [k, k] = [\text{if } i = - \text{ then } k \text{ else } \min(i, k)], i, j
\end{align*}
\]

248
Thus narrowing just discards infinite bounds and makes no improvement on finite bounds, it satisfies the requirements of 9.3.4. According to 9.3.4.4 the system of equations is modified by:

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

The descending approximation sequence is:

\[
\begin{align*}
C_2 &= C_2 \triangleq (C_1 \cup C_4) \\
&= [1, +\infty] \triangleq ([1, 1] \cup [2, 101]) \\
&= [1, +\infty] \triangleq [1, 101] \\
*\ C_2 &= [1, 101] \\
C_3 &= C_2 \cap [1, 101] \\
*\ C_3 &= [1, 101] \cap [+\infty, 100] = [1, 100] \\
\text{stop on that path.} \\
C_5 &= C_2 \cap [101, +\infty] \\
*\ C_5 &= [1, 101] \cap [101, +\infty] = [101, 101] \\
\text{exit.}
\end{align*}
\]

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

9.5 Dual Approximation Methods

The lattice \(\mathcal{A}_{\text{Cont}}\) may be partitioned as follows:

\[
\begin{align*}
X &\geq \text{Int}(X) \\
X &\geq \text{Int}(X) \\
X = \text{Int}(X) \\
\text{AKS} &\rightarrow \text{lp} & \text{gfp} & \text{DRS} & \text{lp} \\
\text{lp} &\rightarrow \text{gfp} & \text{gfp} & \text{lp} & \text{gfp} \\
\text{lp} &\rightarrow \text{gfp} & \text{gfp} & \text{lp} & \text{gfp}
\end{align*}
\]

\text{lp} and \text{gfp} are the least and greatest fixpoints of \(\text{Int}\). The ascending (\text{AKS}) and descending (\text{DRS}) \text{Kleene}'s sequences converge toward \text{lp} and \text{gfp} respectively. These limits are reached when \text{Int} is continuous. When \text{AKS} is infinite we have proposed to use an ascending approximation sequence (\text{AAS}) to approximate \text{lp}. Its limit may be some fixpoint \(\text{lp}'\), or some \(S_m\) such that \(S_m \geq \text{Int}(S_m)\) and \(S_m \geq \text{lp}'\).

When \(X \geq Y\) we have noted \(X \rightarrow \rightarrow \rightarrow \rightarrow Y\).

The truncated descending sequence \(\text{TDS}\) is fundamentally different from \(\text{AAS}\), since it ensures that the successive approximations starting from \(S_m\) remain in the partition \(\{X \mid X \geq \text{Int}(X)\}\), so that their limit \(S'_m\) is greater than \(\text{lp}'\):

\[
\begin{align*}
\text{lp} &\rightarrow \text{gfp} & \text{gfp} & \text{lp} & \text{gfp} \\
\text{lp} &\rightarrow \text{gfp} & \text{gfp} & \text{lp} & \text{gfp}
\end{align*}
\]

It is clear that the ascending approximation sequence \(\text{AAS}\) when starting from \(\text{lp}\) leads to an upper bound of the least fixpoint \(\text{lp}\) of \(\text{Int}\), and the truncated descending sequence \(\text{TDS}\) when starting from \(\text{lp}\) leads to an upper bound of the greatest fixpoint \(\text{gfp}\). Hence the \(\text{AAS}\) and \(\text{TDS}\) methods are not dual, therefore when considering their duals \(\text{DAS}\) and \(\text{TAS}\) we get a means to surround both extreme fixpoints of \(\text{Int}\):

\[
\begin{align*}
\text{lp} &\rightarrow \text{gfp} & \text{gfp} & \text{lp} & \text{gfp} \\
\text{lp} &\rightarrow \text{gfp} & \text{gfp} & \text{lp} & \text{gfp}
\end{align*}
\]
Appendix

We note that \( L \) is a complete \( \sqcup \)-semilattice with partial ordering \( \leq \), supremum \( \sqcup \) and infimum \( \sqcap \). These definitions are given in Birkhoff [61].

Note: \( L \) is a complete lattice.

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

\[
\forall (x, y) \in L^2, \quad (x \leq y) \implies (f(x) \leq f(y))
\]

\[
\Leftrightarrow \forall (x, y) \in L^2, \quad (f(x \sqcup y) = f(x) \sqcup f(y))
\]

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

(\( H2 \)): Let \( F \) be an order-preserving function from the complete semi-lattice \( L, \sqcup, \sqcap, \leq, \sqsupset \) in itself.

(\( L1 \)): The fixed points of \( F \) form a non-empty complete lattice with supremum \( g \) and infimum \( f \) such that:

\[
\forall x, y \in L, \quad x \sqcup y = (x \sqcup f(x)) \sqcap (y \sqcup f(y))
\]

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

We note \( g \) and \( f \) the greatest and least fixed points of \( F \).

(\( H2 \)): Let \( g \) and \( f \) be such that:

\[
(\forall x) F(x) = x \implies (\forall x) x = y(x)
\]

(\( H2 \)): Let \( F \) be an order-preserving function from the complete semi-lattice \( L, \sqcup, \leq, \sqcap \) in itself.

(\( H3 \)): Let \( L \) be an order-preserving function from the complete semi-lattice \( L, \sqcup, \sqcap, \leq, \sqsupset \) in itself.

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

(\( L3 \)): Let \( F : L \to L \) be an order-preserving function from the semi-lattice \( L, \sqcup, \leq, \sqcap \) in itself, \( f \) and \( g \) respectively the least and greatest fixed points of \( F \), then:

\[
\forall x \in L, \quad (g \sqcup F(x) \geq x) \implies (g \geq x)
\]

(\( L3 \)): Let \( F : L \to L \) be an order-preserving function from the semi-lattice \( L, \sqcup, \leq, \sqcap, \sqsupset \) in itself, \( f \) and \( g \) respectively the least and greatest fixed points of \( F \), then:

\[
\forall x \in L, \quad (f \sqcap F(x) \leq x) \implies (f \leq x)
\]
(T1): H1, H1, H2, H3 imply that the greatest fixpoints \( g \) and \( \alpha \) of \( F \) and \( \bar{F} \) are related by:
\[
\{ \alpha(g) \leq \bar{g} \} \quad \text{and} \quad \{ g \geq \gamma(\bar{g}) \}
\]
Proof:
The existence of \( g \) and \( \bar{g} \) is stated by (L1).
\[
\begin{align*}
g = \alpha(\bar{g}) & \geq \alpha(g) \\
\bar{g} = \alpha(F(g)) & \geq \alpha(g) \\
\bar{g} = \alpha(F(\alpha(g))) & \geq \alpha(g) \\
\end{align*}
\]
trivially
since \( \bar{g} = F(g) \)
\( \alpha \) is monotone, \( \leq \) transitive
\( g \geq \alpha(g) \)
L3
\( \gamma(g) \geq \gamma(\alpha(g)) \)
H2.4
\( \gamma(F) \geq g \)
H2.6, \( \leq \) transitive.
Q.E.D.

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

(T2): H1, H1, H2, H3 imply that the least fixpoints \( \ell \) and \( \bar{\ell} \) of \( F \) and \( \bar{F} \) are related by:
\[
\{ \gamma(\bar{\ell}) \geq \ell \} \quad \text{and} \quad \{ \bar{\ell} \leq \alpha(\ell) \}
\]
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 \( \leq \)) 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 \( f(\bigcup \{ x \mid x \in X \}) = \bigcup \{ f(x) \mid x \in X \} \).

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

(H4'): Let \( \bar{F} \) be a continuous function from the complete semi-lattice \( \langle L, \geq, \alpha, \gamma \rangle \) in itself.

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

(L4): H4(H4') implies that \( F(\bar{F}) \) has a least fixpoint \( \ell(\bar{F}) \) which is the limit \( \bigcup \{ F^i(\ell) \} \) of the Kleene's sequence \( \ell \leq F(\ell) \leq \ldots \leq F^n(\ell) \)
\( \leq \ldots \)
(The proof is easy to adapt from Kleene[52]'s proof of the first recursion theorem pp. 348-349).