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
ABSTRACT INTERPRETATION: A UNIFIED LATTICE MODEL FOR STATIC ANALYSIS
OF PROGRAMS BY CONSTRUCTION OR APPROXIMATION OF FIXPOINTS

Patrick Cousot* and Radhia Cousot**
Laboratoire d'Informatique, U.S.M.G., BP. 53
38041 Grenoble cedex, France

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 \(\{(+), (-), (\#)\}\) where the semantics of arithmetic operators is defined by the rule of signs. The abstract execution \(-1515 * 17 \rightarrow (-) * (+) \rightarrow (-) * (+) \rightarrow (-)\), proves that -1515 * 17 is a negative number. Abstract interpretation is concerned by a particular underlying structure of the usual universe of computations (the sign, in our example). It gives a summary of some facets of the actual executions of a program.

In general this summary is simple to obtain but inaccurate (e.g., \(-1515 + 17 \rightarrow (+) + (+) \rightarrow\).

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
3.2 Semantics of Programs

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

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

- 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:
  \[ Env = \text{Ident}^\geq \rightarrow \text{Values} \]

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

  In particular the projection \( \text{val} \upharpoonright \text{Bexpr} \upharpoonright \) of the function \( \text{val} \upharpoonright \text{domain} \text{Bexpr} \) has the functionality:
  \[ \text{val} \upharpoonright \text{Bexpr} \upharpoonright [\text{Env} \rightarrow \text{Bool}] \]

- The state set "States" consists of the set of all information configurations that can occur during computations:
  \[ \text{States} = \text{States}^0 \times \text{Env} \]

  A state \( (s \in \text{States}) \) consists of a control state \( (\text{cs}(s)) \) and an environment \( (\text{env}(s)) \), such that:
  \[ (s \in \text{States}) \Rightarrow (\text{cs}(s), \text{env}(s)) \]

  We use a continuous conditional function \( \text{cond}(b, \{e_1, e_2\}) \) equal to \( b \rightarrow e_1 \lor e_2 \lor \top \) respectively as the value of \( b \) is \( i, \text{true}, \text{false} \) 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):
  \[ n\text{-state} : \text{States} \rightarrow \text{States} \]

\[ n\text{-state}(s) = \]

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

- Let \( \bot\text{-Env} \) be the bottom function on \( \text{Env} \) such that:
  \[ (s \in \text{Ident}^\geq \Rightarrow \text{Env}(s) = \bot\text{-Values}) \]

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

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
$$n\text{-state}^\infty = \nu F. (n\text{-state} \circ F)$$
where $\nu F(f)$ 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 $Cq$ at some program point $q \in \text{Arcs}$ of a program $P$ to be the set of all environments which may be associated to $q$ in all the possible computation sequences of $P$:
$$Cq \in \text{Contexts} = \mathcal{E}_{\text{Env}}$$
$$Cq = \{e \mid (\exists n \geq 0, i_n \in \text{I-states} \mid <q,e> = n\text{-state}^n(i_n))\}$$
The context vector is a context to each of the program points of a program:
$$Cv \in \text{Context-Vectors} = \text{Arcs}^0 \rightarrow \text{Contexts}$$
$$Cv = \lambda q. (\{e \mid (\exists n \geq 0, i_n \in \text{I-states} \mid <q,e> = n\text{-state}^n(i_n))\})$$

According to the semantics of programs, the context $Cv(r)$ associated to arc $r$ is related to the context $Cv(q)$ at arc $q$ adjacent to $r$,
$$(\exists q \in \text{Origin}(r), \frac{q\rightarrow r}{Cv(r)}).$$
From the definition of the state transition function we can prove the equation:
$$Cv(r) = n\text{-context}(r, Cv)$$
$$\text{n-context : Arcs}^0 \times \text{Context-Vectors} \rightarrow \text{Contexts}$$
is defined by:
$$\text{n-context}(r, Cv) = \begin{cases} \text{origin}(r) \in \begin{cases} \text{Entries} = \{\text{Env}\} \cup \text{Assigns} \cup \text{Tests} \cup \text{Junctions} \subset \mathcal{E}_{\text{Env}} \end{cases} \end{cases}$$
$$\text{env-on}(r) = \text{n-context}(\text{env}(r)) $$
$$(\text{Cond}(r = \text{eq}(x), \text{env}(s), \emptyset)), \text{eq}$$

We define $\text{env-on} : \text{Arcs}^0 \rightarrow \{\text{States} \rightarrow \mathcal{E}_{\text{Env}}\}$ to be
$$\lambda r. (\text{Cond}(r = \text{eq}(x), \text{env}(s), \emptyset)).$$

Since the equation $Cv(r) = n\text{-context}(r, Cv)$ must be valid for each arc, $Cv$ is a solution to the system of "forward" equations:
$$Cv = \text{F-cont}(Cv)$$
where
$$\text{F-cont} : \text{Context-Vectors} \rightarrow \text{Context-Vectors}$$
is defined by:
$$\text{F-cont}(Cv) = kr \cdot n\text{-context}(r, Cv)$$
$\text{Context-Vectors}$ is a complete lattice with union $\bigvee$ such that $Cv \uplus Cv_2 = \lambda r. (Cv_1(r) \uplus Cv_2(r))$.

$\text{F-cont}$ is order preserving for the ordering $\preceq$ of Context-Vectors which is defined by:
$$Cv_1 \preceq Cv_2 \iff (\forall r \in \text{Arcs}, Cv_1(r) \preceq Cv_2(r))$$

Hence it is known that $\text{F-cont}$ has fixpoints, Tarski [55]. However, it is trivial to exhibit examples which show that these fixpoints are not always unique. Fortunately, it can be shown that $Cv$ is included in any solution $\mathcal{G}$ to the system of equations $X = \text{F-cont}(X), (Cv \preceq \mathcal{G})$. Tarski [55] shows that this property uniquely determines $Cv$ as the least fixpoint of $\text{F-cont}$. Thus $Cv$ can be equivalently defined by:

$$D1 : Cv = \lambda q. (\{e \mid (\exists n \geq 0, i_n \in \text{I-states} \mid <q,e> = n\text{-state}^n(i_n))\})$$

or

$$D2 : Cv = \nu F. (\lambda q. (\{e \mid (\exists n \geq 0, i_n \in \text{I-states} \mid <q,e> = n\text{-state}^n(i_n))\}))$$

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

(a) $Cv(q)$ contains at least the environments $e$ which may be associated to $q$ during any execution of $P$,

(b) $Cv(q)$ contains only the environments $e$ which may be associated to $q$ during an execution of $P$,

$Cv$ is merely a static summary of the possible executions of the program. However, our definitions $D1$ or $D2$ of $Cv$ cannot be utilized at compile time since the computation of $Cv$ consists in fact in running the program (for all the possible input data). In practice compilers may consider states which can never occur during program execution (e.g. some compilers consider that any program always performs 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
$$I = \langle A\text{-Cont}, \preceq, \leq, \top, \bot, \text{Int}\rangle$$
where the set of abstract contexts is a complete $\preceq$-semilattice with ordering $\preceq$, $\langle x \preceq y \iff (x \preceq y) \iff (x \leq y) \iff (x \preceq y)\rangle$. This implies that $A\text{-Cont}$ has a supremum $\top$. We suppose also $A\text{-Cont}$ to have an infimum $\bot$. 240
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 \( A-Cont = \text{Arcs}^9 \rightarrow A-Cont \). Whatever \( (Cv', Cv'') \in A-Cont^2 \) may be, we define:

\[
Cv' \sqsupset Cv'' = \lambda r. Cv'(r) \circ Cv''(r)
\]

\[
Cv' \sqsubseteq Cv'' = \{ r \in \text{Arcs}^9 \mid Cv'(r) \leq Cv''(r) \}
\]

\[
\sim = \lambda r. \top \quad \text{and} \quad \perp = \lambda r. \bot
\]

\( A-Cont, \sim, \leq, \top, \bot \) can be shown to be a complete lattice. The function:

\[
\text{Int} : \text{Arcs}^9 \times A-Cont \rightarrow A-Cont
\]

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 \( \text{arc} r \) of \( n \) \( (r \in a-\text{succ}(n)) \) is equal to \( \text{Int}(r, C) \). \( \text{Int} \) is supposed to be order-preserving:

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

\[
\{ Cv' \sqsubseteq Cv'' \} \Rightarrow \{ \text{Int}(a, Cv') \leq \text{Int}(a, Cv'') \}
\]

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:

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

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

5.2 Typology of Abstract Interpretations

The restriction that "A-Cont" must be a complete lattice is not drastic since Mac Neille[37] showed that any partially ordered set \( S \) can be embedded in a complete lattice so that inclusion is preserved, together with all greatest lower bounds and largest upper bounds existing in \( S \). Hence in practice the set of abstract contexts will be a lattice, which can be considered as a join \( (\cup) \) semi-lattice or a meet \( (\cap) \) semi-lattice, thus giving rise to two dual abstract interpretations.

It is a pure coincidence that in most examples (see 5.3.2) the \( \cap \) or \( \cup \) operator represents the effect of path convergence. The real need for this operation is

Examples:

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

5.3 Examples

5.3.1 Static Semantics of Programs

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

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

where \( \text{Contexts} \), \( u, \leq, \text{Env} \), \( n-\text{context} \), \( \text{Context-Vectors} \), \( \leq \), \( E-\text{Cont} \) respectively correspond to \( A-Cont, \sim, \leq, \top, \bot, \text{Int}, A-Cont, \sim, \leq, \text{Int} \).

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 \( \text{arc} r \), if whenever control reaches \( r \), the value of the expression has been previously computed, and since the last computation of the expression, no argument of the expression has had its value changed.

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

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

\( B-\text{vect} \) is clearly a complete boolean lattice. The interpretation of basic nodes is defined by:

\[
\text{Int}(n, B) = B
\]
The determination of available expressions, back-
dominators, intervals, ... requires a forward sys-
 tem of equations. Some global flow problems, nota-
 bly 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 a particular solver C
 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-
 rence 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 \( \mathcal{I} = \langle A\text{-Cont}, \gamma, \bar{z}, T, i, \text{Int} \rangle \) of a program is consistent with a "concrete" interpretation \( \mathcal{I} = \langle C\text{-Cont}, \gamma, \bar{z}, T, i, \text{Int} \rangle \) if the context vector \( \bar{C} \) resulting from \( \gamma \) is a cor-
 rect approximation of the particular solver \( C \) resulting
 from the more refined interpretation \( \mathcal{I} \). This
 may be rigorously defined by establishing a corre-
 spondence (\( \alpha : \text{abstraction} \)) between concrete and ab-
 stract context vectors, and inversely (\( \gamma : \text{concret-
 ization} \)) and requiring:

\[
\begin{align*}
\forall \bar{c} \in A\text{-Cont}, & \quad \bar{c} = \gamma(\alpha(\bar{c})) \\
\forall \bar{c} \in C\text{-Cont}, & \quad \bar{c} = \alpha(\gamma(\bar{c}))
\end{align*}
\]

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

If \( \delta : \mathbb{D} \rightarrow \mathbb{D}' \) we note \( \bar{D} = A\text{-Cont} \circ \mathbb{D} \) and \( \bar{D}' = A\text{-Cont} \circ \mathbb{D}' \) and \( \bar{\mathcal{I}} : \bar{D} \rightarrow \bar{D}' = \lambda \bar{d}. (\lambda r. f(d(r))) \).

We will suppose \( \alpha \) and \( \gamma \) to satisfy the following hypothesis:

\[
\begin{align*}
6.1 & \quad \alpha : C\text{-Cont} \	o A\text{-Cont} \\
6.2 & \quad \alpha \text{ and } \gamma \text{ are order-preserving} \\
6.3 & \quad \forall \bar{C} \in A\text{-Cont}, \bar{C} = \alpha(\gamma(\bar{C})) \\
6.4 & \quad \forall \bar{C} \in C\text{-Cont}, \bar{C} = \gamma(\alpha(\bar{C}))
\end{align*}
\]

Intuitively, hypothesis 6.2 is necessary because
context inclusion (that is property comparison)
 must be preserved by the abstraction or concreti-
 zation 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 \( \gamma(\alpha(C)) \supseteq C \). 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 con-
 structs:

\[
\begin{align*}
\forall (a, \bar{x}) \in A\text{-Cont}, & \quad \gamma(\text{Int}(a, \bar{x})) \supseteq \text{Int}(a, \gamma(\bar{x})) \\
6.5 & \quad \forall (a, \bar{x}) \in C\text{-Cont}, \\
& \quad \text{Int}(a, \alpha(\bar{x})) \supseteq \alpha(\text{Int}(a, \bar{x}))
\end{align*}
\]

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

\[
\begin{align*}
\alpha & \quad \gamma \\
\text{Int}(a, \bar{x}) & \quad \text{Int}(a, \gamma(\bar{x}))
\end{align*}
\]

Suppose we want to compute the concrete output con-
text \( C_0 \) (associated with \( a \)) resulting from con-
crete input contexts \( C_1 \) : \( C_0 = \text{Int}(a, C_1) \). We can
as well approximate this computation in the abstract
universe, and get \( C_0' = \gamma(\text{Int}(a, \alpha(\bar{C}_1))) \). 6.5 requires
\( C_0' \) to contain at least \( C_0 \), that is \( C_0' \subseteq C_0 \). On the
contrary we do not require \( C_0' \) to contain at most
\( C_0 \), that is \( C_0 \subseteq C_0' \) is not compulsory.

We will say that \( \mathcal{I} \) is a refinement of \( \mathcal{I} \), or that
\( \mathcal{I} \) is an abstraction of \( \mathcal{I} \), denoted \( I \leq (\alpha, \gamma) \mathcal{I} \),
if and only if there exist \( \alpha \) and \( \gamma \) satisfying hypothe-
sis 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 pre-
mitive language constructs (6.5). Theorems T1 and
T2 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

\[
I_{\mathcal{S}} = \langle \text{Contexts}, \bar{z}, \text{Env}, \emptyset, a\text{-context} \rangle
\]

any abstract interpretation \( \mathcal{I} \) of \( P \), consistent with
\( I_{\mathcal{S}} \), \( I_{\mathcal{S}} \leq (\alpha, \gamma) \mathcal{I} \) is consistent with the seman-
tics of \( P \), which implies:

\[
\begin{align*}
\forall \bar{q} \in A\text{-Cont}, & \quad \bar{q} = \gamma(\alpha(\bar{q})) \\
6.3 & \quad \forall (a, \bar{x}) \in C\text{-Cont}, \\
& \quad \text{Int}(a, \alpha(\bar{x})) \supseteq \alpha(\text{Int}(a, \bar{x}))
\end{align*}
\]

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 \( \mathcal{P}(x_1, \ldots, x_n) \)
\( \in \text{Pred} \) 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:

\[
I_{\mathcal{S}} = \langle \text{Pred}, \lor, \Rightarrow, \text{true}, \text{false}, \text{n-pred} \rangle
\]

242
where \( n\text{-pred} \) defines Floyd[67]'s strongest post condition:

\[
\begin{align*}
n\text{-pred}(r, P v) &= \text{let}(s = \text{origin}(r), (p \text{ be } a\text{-pred}(\text{origin}(r)))(r) \text{ within case } n \text{ in }
\begin{align*}
\text{Entries} &\quad \Rightarrow (\forall x < \text{Ident}, x = i\text{-Values}) \\
\text{Junctions} &\quad \Rightarrow \quad \text{qact}(\text{p(q)}) \\
\text{Tests} &\quad \Rightarrow \text{case } n \in \\
&\quad (\text{a-succ}\text{-}(n)) \Rightarrow P v(p) \quad \text{and} \quad \text{test}(n) \\
&\quad (\text{a-succ}\text{-}(n)) \Rightarrow P v(p) \quad \text{and} \quad \text{not test}(n) \\
\text{esac}
\end{align*}
\end{align*}
\]

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

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

\[
\begin{align*}
\alpha : \text{Contexts} &\rightarrow \text{Pred} \\
= \alpha C &\rightarrow (x \text{ and } (x = e(x))) \\
&\in C \times \text{Ident} \\
\gamma : \text{Pred} &\rightarrow \text{Contexts} \\
= \lambda P . \{e \mid P(e(x))/x, x \in \text{Ident}\}
\end{align*}
\]

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

\[
\{\psi \in \text{Arcs}, \forall P v \in \frac{\text{Pred}}{\quad \alpha(\text{context}(a, \gamma(P v))) \Rightarrow n\text{-pred}(a, P v)}
\]

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

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

\[
\forall q \in \text{Arcs}, \exists P v \text{ be the result of } I_{\text{gs}}' \quad \{X \geq 0, \exists i_{8} \in 1\text{-states} | \langle q, e \rangle = n\text{-state}_{8}(i_{8}) \} \Rightarrow (P v(q) \Rightarrow \alpha(e))
\]

7. The Lattices of Abstract Interpretations

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

\[
\begin{align*}
\text{reflexive} &\quad : (I \leq (1, 1)) \text{ where } \gamma = \lambda x . x \text{ is the identity function, } \\
\text{transitive} &\quad : (I \leq (\alpha, \gamma_{1}), I' \leq (\alpha_{2}, \gamma_{2})) \text{ imply } I \leq (\alpha_{1} \circ \alpha_{2}, \gamma_{2} \circ \gamma_{1}), I'
\end{align*}
\]

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

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

is an equivalence relation. We have:

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

The proof gives some insight in the abstraction process:

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

2. - reciprocally,

If \( I \leq (\alpha, \gamma_{1} I') \), let \( \equiv (\alpha, \gamma_{1}) \) be the equivalence relation defined on \( I \) (properly speaking, on the set of abstract contexts of \( I \)) by:

\[
\{ x \in (\alpha_{1} y) \} \Leftrightarrow \{ x \in (\alpha_{1} y) \}
\]

\( \forall x' \in I' \), each equivalence class \( C_{x'} = \{ x \in I \mid x' = (\alpha_{1} y) \} \) has a least upper bound which is \( \gamma_{1}(x') \). Hence the projection \( \alpha_{1} \mid \gamma_{1}(I') \) of \( \alpha_{1} \) on \( \gamma_{1}(I') \) is a bijection from the set \( \gamma_{1}(I') \) of 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 bijective:

\[
\gamma_{1}(I') \text{ and } (\alpha_{2}, \gamma_{2}) I' \text{ are bijections, hence } \forall x' \in I', \exists x \in I \text{ such that } x' = (\alpha_{1} \gamma_{1}(I'))(x). \text{ Likewise, } x \in \gamma_{1}(I') \Rightarrow x' = (\alpha_{1} \gamma_{1}(I'))(x) \}
\]

Therefore, \( \forall x' \in I' \), \( \exists x \in I \) such that \( x' = (\alpha_{1} \gamma_{1}(I'))(x) \). Thus \( (\alpha_{1} \gamma_{1}(I')) = (\alpha_{2} \gamma_{2}(I')) \). Hence \( \alpha_{1} \mid \gamma_{1}(I') \) is a bijection between \( \gamma_{2}(I) \) and \( I' \). Since \( \alpha_{1} \gamma_{1}(I') \) is a bijection between \( I \) and \( \gamma_{2}(I) \), the composition

\[
(\alpha_{1} \mid \gamma_{1}(I')) \circ (\alpha_{2} \mid \gamma_{2}(I')) = (\alpha_{1} \mid \gamma_{1}(I')) = (\alpha_{2} \mid \gamma_{2}(I'))
\]

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

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

In particular, we can restrict I to be set of interpretations which abstract \( I_{\text{gs}} \). I is then a lattice, (with ordering \( \Sigma \)) 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 \( \alpha(S) = (-\infty, +\infty) \).

The abstract contexts are then, (Cousot[76]):
8. Abstract Evaluation of Programs

The system of equations:
\[ Cy : \text{Int}(Cv) \]
resulting from an interpretation \( I = \langle \text{A-Cont}, \cdot, \leq, I, t, i, \text{Int} \rangle \) 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):
\[ Cy := (C := I; \text{until } C = \text{Int}(C) \text{ do } C := \text{Int}(C) \text{ repeat}; C) \]

8.1 Correctness

If \( \text{Int} \) is supposed to be a complete morphism (i.e. infinitely distributive over \( \cdot \)) then \( Cy \) 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 \( Cy \) 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 isorome, \( Cy \) is an approximation (\( ? \)) of the least fixpoint (e.g. Knu and Ullman[75]).

8.2 Termination

The abstract evaluation terminates if the Kleene’s sequence is finite. This may be the case because \( \text{A-Cont} \) is finite (e.g. type checking in ALGOL 60, Naur[65]), or a finite subset only is to be considered for any particular program (e.g. type checking in ALGOL 68), or \( \text{A-Cont} \) may be of finite length \( m \) (the length of any strictly increasing chain is bounded by \( m \), Kildall[73], Wegbreit[75]) or \( \text{A-Cont} \) may satisfy the ascending chain condition (every strictly increasing chain is finite, although not bounded). A lattice may have infinite chains, although \( \text{Int} \) is chosen so that 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 (Sintoff[75]) or heuristics (Katz and Manual[76]) may be used to pass to the limit, or approximate it, (Coons[76]).

8.3 Efficiency

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

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

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

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

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

$$\text{Kir}(r, Cv) = \begin{cases} \text{let } n \text{ be origin}(r) \text{ within } \text{case } n \text{ in } \text{Entries} & \rightarrow 1 \text{ \{unique entry node\}} \\ \text{case } r \text{ in } \{a\text{-succ}(n)\} & \rightarrow \text{Cv}(a\text{-pred}(n)) \\ \text{case } r \text{ in } \{a\text{-succ}(n)\} & \rightarrow \text{Cv}(a\text{-pred}(n)) \ast \frac{\text{Prob}(\text{test}(n) = \text{true})}{\text{Prob}(\text{test}(n) = \text{false})} \\ \text{esac} \end{cases}$$

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

$$\text{Cv}_0 \leq \text{Cv}_1 \leq \ldots \leq \text{Cv}_n \leq \ldots$$

then

$$\max_{n \geq 0} \left( \sum \frac{\text{Cv}_i(p)}{\text{Cv}_i((p))} \right) = \sum \frac{\text{Cv}_i(p)}{\text{Cv}_i((p))}$$

and

$$\max_{n \geq 0} \left( \sum \frac{n_i \ast q}{\text{Cv}_i((p))} \right) = \sum \frac{n_i \ast q}{\text{Cv}_i((p))} \ast q.$$

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

- Let $P$ be the program "begin L: go to L end". The number $n$ of iterations in the loop is given by the minimal solution to the equation $n = n + 1$ which is limit of $0 + 1 + 1 + 1 + \ldots$

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

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

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 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 $\tilde{I}$ ($1 \leq \tilde{I}$) may be used for that purpose (e.g. Ternenbaum[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 $\text{Cv}$ of $\text{Int}$ is unreachable, we look for an upper bound $\text{Ub}$ of $\text{Cv}$, since according to the correctness requirement $6.0, \text{Cv} \leq \gamma(\text{Cv})$ and $\text{Cv} \leq \text{Ub}$ implies $\text{Cv} \leq \gamma(\text{Ub})$.

9.1.1 Increasing Approximation Sequence

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

9.1.1.1 ($\forall n \geq 0$, $C = \text{A-Int}^n(\tilde{C})$ and not($\text{Int}(C) \nleq C$))

$$\text{Cv} \leq \gamma(\text{Cv}) \nleq \text{Ub} \implies \text{Cv} \leq \gamma(\text{Ub})$$

9.1.1.2 Every infinite sequence $S_0, S_1, S_2, \ldots$ is not strictly increasing.

The approximation sequence $S_0, S_1, S_2, \ldots$ is recursively defined by:

9.1.1.3 $S_0 = 1$

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

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

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

Let $n$ be the least natural number (eventually infinite) such that $S_m = S_{m+1}$. We set $S_k \nleq S_k$. We know from 9.1.1.3 that not($\text{Int}(S_k) \nleq S_k$). Whence by definition of the ordering $S_k \neq \text{Int}((S_k) \nleq S_k$. Since $S_k \nleq \text{Int}(S_k) \nleq S_k$ is always true, we can state that $S_k = \text{Int}(S_k) \nleq S_k$. Besides not($\text{Int}(S_k) \nleq S_k$) and 9.1.1.1 imply:

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

and therefore we conclude $S_{k+1} \nleq S_k$. We set $1 \leq m$. Moreover 9.1.1.2 implies that $m$ is finite. Q.E.D.

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

$$\forall X \in A\text{-Cont}, (\text{Int}(X) \nleq X) \implies (\text{Cv} \nleq X)$$

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

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

9.1.3 Widening in Increasing Approximation sequences

The definition of the approximate interpretation $A$-Int in 9.1.1 is global. We now indicate a way to construct $\bar{A}$-Int by local modifications to Int.

Let $(q, r) \in \text{Arcs}^2$, we say that the context associated to $q$ is dependent on the context associated to $r$, if and only if:

$$\exists \sigma \in A \text{-Cont}, \exists c \in A \text{-Cont} \mid \text{Int}(q, Cv \neq \text{Int}(q, Cc \cup c))$$

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

In any infinite strictly increasing Kleene's sequence $Cv_1, \ldots, Cv_m, \ldots$ since Arcs is finite there is some arc $q$ for which the sequence $Cv_1(q), \ldots, Cv_m(q), \ldots$ 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 $\triangledown$ called widening defined by:

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

9.1.3.2 $\forall (C, C') \in A \text{-Cont}^2, C \circ C' \subseteq C \triangledown C'$

9.1.3.3 Every infinite sequence $s_0, \ldots, s_n, \ldots$ of the form $s_0 = C_0, \ldots, s_n = C_n, \ldots$ (where $C_0, \ldots, C_n, \ldots$ are arbitrary abstract contexts) is not strictly increasing.

- The set $W$-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 $Cv = \text{Int}(Cv)$ contains at least a widening arc:

$$\forall n \in \mathbb{N}, q_n \in W \text{-arcs}. \quad (e.g. \text{in a forward})$$

As before, we define:

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

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

Let us consider a sequence $S_0 = \bar{S}_0, \ldots, S_m = \bar{S}_m$.

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

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

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

An infinite strict increasing sequence $S_0 = \bar{S}_0, \ldots, S_n = \bar{S}_n, \ldots$ cannot be strictly increasing since otherwise there would exist some widening arc $q$ for which the sequence $S_0(q), \ldots, S_n(q), \ldots$ would never stabilize thus contradicting 9.1.3.3.

We now prove 9.1.1.1 that is to say that:

$$\forall n \in \mathbb{N}, s_n \equiv \bar{A} \text{-Int}(\bar{S}_n)$$

implies $A \text{-Int}(\bar{S}_n) \subseteq \bar{A} \text{-Int}(\bar{S}_n)$.
Let us note \([a, b]\) where \(a \leq b\) the predicate \(a \leq x \leq b\). The system of equations corresponding to the example is:

1. \(C0 = [1, 100]\)
2. \(C1 = [1, 1]\)
3. \(C2 = C1 \cup C4\)
4. \(C3 = C2 \cap [-\infty, 100]\)
5. \(C4 = C3 + [1, 1]\)
6. \(C5 = [1, 100] + [1, 1]\)
7. \(C6 = [2, 101]\)

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

\(C5 = C2 \cap [101, \infty]\)
\(= [1, \infty] \cap [101, \infty]\)
\(C6 = [101, \infty]\)
exit, stop.

The final context on each arc is marked by a star '*'. Note that the results are approximate ones, (e.g. \(C5\)).

In this example the widening is a very rough operation which introduces a great loss of information. However it can be seen in the trace that tests behave like filters. Furthermore, for PASCAL like languages, one can first use the bounds given in the declaration of \(x\) before widening to infinite limits.

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

The fixpoint approximation procedure leads to an...
Let $D$ be a finite decreasing sequence by imposing an arbitrary upper bound on $n$.

Let $D$ be a finite decreasing sequence by imposing an arbitrary upper bound on $n$.

Let $D$ be a finite decreasing sequence by imposing an arbitrary upper bound on $n$.

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

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

Let us 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 $S'_p = S'_p$. Trivially from 9.1.1:

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

If $p > 0$ then $S'_0 \neq \text{Int}(S'_0)$, 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:

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

Since $S'_{k-1}$ is order preserving we have:

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

By transitivity $S'_k \geq \text{Int}(S'_{k-1})$ and since 9.3.2.3 implies $S'_k \neq \text{Int}(S'_{k-1})$ we have from 9.3.2.1:

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

Since 9.3.2.3 implies $S'_k \neq D - \text{Int}(S'_{k-1})$ we have:

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

By recurrence on $k$ the result is true for $k \leq p$. Moreover 9.3.2.2 implies that $p$ is finite. Q.E.D.

9.3.3 Generalization of Elkmene's Decreasing Sequence

When $A$ satisfies the descending chain condition, one can choose $D$ 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 $CV$ of $\text{Int}$.

The limit of the decreasing sequence $S'_0 = \sim, \ldots, S'_p = \text{Int}(S'_p), \ldots$ is an upper bound of the greatest fixpoint 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 $D$ by local modifications to $\text{Int}$:

9.3.4.1 $\Delta : A \times A \rightarrow A$

9.3.4.2 $\forall (C, C') \in A \times A$, $C \geq C'$, $C \geq C'$ implies $C \geq C'$.

9.3.4.3 Every infinite sequence $S_q, \ldots, S_n$ of the form $S_q = C_0, S_{q+1} = S_q \Delta C_1, \ldots, S_n = S_{n-1} \Delta C_n$, for arbitrary abstract contexts $C_0, \ldots, C_n$, is not strictly decreasing.

The approximate interpretation $D$ is defined by:

9.3.4.4 $D - \text{Int} = \lambda(q, CV).$ if $q \in W$-arcs then $CV(q) \geq D - \text{Int}(q, CV)$

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 $CV(q) \geq D - \text{Int}(q, CV)$ implies $CV(q) \geq D - \text{Int}(q, CV)$.

The proof of termination is very similar to the one outlined for $A$ 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:

(1) $C_1 = [1, 1]$
(2) $C_2 = C_1 \cup C_4$
(3) $C_3 = C_2 \cap [-∞, 100]$
(4) $C_4 = C_3 + [1, 1]$
(5) $C_5 = C_2 \cap [100, +∞]$

The ascending approximation sequence led to the approximate solution:

* $C_1 = [1, 1]$
* $C_2 = [1, +∞]$
* $C_3 = [1, 100]$
* $C_4 = [2, 101]$
* $C_5 = [101, +∞]$

Let us define the narrowing $\Delta$ of intervals by:

- $[i, j] \Delta [k, l] = [\text{if } i = -∞ \text{ then } k \text{ else } \min(i, k), \text{ if } j = +∞ \text{ then } k \text{ else } \max(j, l)]$
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) \( C_2 = C_2 \cap (C_1 \cup C_4) \)

The descending approximation sequence is:

\[
\begin{align*}
C_2 &= C_2 \cap (C_1 \cup C_4) \\
    &= [1, +\infty) \cap ([1, 1] \cup [2, 101]) \\
    &= [1, +\infty) \cap [1, 101] \\
C_3 &= C_2 \cap [-\infty, 100] \\
C_4 &= C_3 \cap [1, 101] \cap [100] = [1, 100] \\
C_5 &= C_2 \cap [101, +\infty] \\
C_6 &= [1, 101] \cap [101, +\infty] = [101, 101]
\end{align*}
\]

stop on that path.

On that example the approximate solution has been improved so that the laser endpoint is reached but...
Any of the AAS, TDS, DAS, TAS methods may yield a fixpoint \( fp \) which is not the fixpoint \( lfp \) or \( gfp \) of interest. None of these methods can improve \( fp \) to reach \( lfp \) or \( gfp \), therefore a "fixpoint improvement method" is necessary. It is our feeling that such a method could be designed only when considering that A-Gont possesses a richer structure (i.e. for particular applications).

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

\[ \text{Acknowledgments} \]

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.)


Henderson[75]. Finite state modelling in program


Scott[71]. The lattice of flow diagrams. Symp. on Semantics of Programming Languages. Springer-Verlag Lecture Notes in Math. (E. Engeler, ed.), Vol. 188.


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


13. Appendix

We note that L is a complete u-semilattice with partial ordering ≤, supremum τ and infimum 1. These definitions are given in Birkhoff[61].

Note: L is a complete lattice.

(proof in Birkhoff[61], p. 49).

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

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

⇐⇒ {∀(x, y) ∈ L², [f(x) ∨ y) ≥ f(x) ∨ f(y)]}

(H1): Let F be an order-preserving function from the complete semi-lattice <L, u, ≤, τ, 1> in itself.

(H2): Let F be an order-preserving function from the complete semi-lattice <L, u, ≤, τ, 1> in itself.

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

F = α(F(x))

(x ≤ F(x))

(L2): [H3.1] ⇐⇒ [H3.2]

Proof:

∀x ∈ L,

F(α(F(x))) ≥ α(F(x)) for H3.1 and transitivity.

F(F(x)) ≥ F(F(x)) for H2.4

γ(F(x)) ≥ F(γ(x)) for H2.6 and transitivity.

∀x ∈ L,

γ(F(x)) ≥ F(γ(x)) for H2.3

α(F(x)) ≥ F(α(F(x))) for 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 → L be an order-preserving function from the semi-lattice <L, u, ≤, τ, 1> in itself, and g respectively the least and greatest fixpoints of F, then:

∀x ∈ L, {F(x) ≥ x} ⇐⇒ {g ≥ x}

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

∀x ∈ L, {x ∈ F(x) ≤ x} ⇐⇒ {x ≤ x}
(T1) \( H_1, H_1, H_2, H_3 \) imply that the greatest fixpoints \( g \) and \( \bar{g} \) of \( F \) and \( \bar{F} \) are related by:

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

Proof:

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

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

Q.E.D.

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

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

\[
(\gamma(\bar{\ell}) \preceq \ell) \quad \text{and} \quad (\bar{\ell} \preceq \alpha(\ell))
\]

According to Scott[7]) a subset \( X \subseteq L \) is called directed if every finite subset of \( X \) has an upper bound (in the sense of \( \preceq \)) belonging to \( X \). (An obvious example of a directed subset is a non-empty ascending chain). A function \( f : D \to D \) is called continuous if whenever \( X \preceq L \) 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 \( \langle L, \cup, \preceq, 1, \rangle \) in itself.

(H4'): Let \( \bar{F} \) be a continuous function from the complete semi-lattice \( \langle L, \bar{\cup}, \bar{\preceq}, 1, \rangle \) in itself.

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

(L4): \( H_4(\bar{H}_4) \) implies that \( F(\bar{F}) \) has a least fix-point \( \ell(\bar{F}) \) which is the limit \( \cup F^i(1) \) 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).