ABSTRACT INTERPRETATION: A UNIFIED LATTICE MODEL FOR STATIC ANALYSIS
OF PROGRAMS BY CONSTRUCTION OR APPROXIMATION OF FIXPOINTS
Patrick Cousot* and Radha 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 notation 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 \(-1515 \times 17\) may be understood to denote computations on the abstract universe \((+, -, \times)\) where the semantics of arithmetic operators is defined by the rule of signs. The abstract execution \(-1515 \times 17 \Rightarrow -(+) \times (+) \Rightarrow (-) \times (+) \Rightarrow (-)\), proves that \(-1515 \times 17\) is a negative number. Abstract interpretation is concerned by a particular underlying structure of the usual universe of computations (the sign, in our example). It gives a summary of some facets of the actual executions of a program. In general this summary is simple to obtain but inaccurate (e.g. \(-1515 \times 17 \Rightarrow -(+) \times (+) \Rightarrow (-) \times (+) \Rightarrow (\cdot)\)). Despite its fundamentally incomplete results abstract interpretation allows the programmer or the compiler to answer questions which do not need full knowledge of program executions or which tolerate an imprecise answer, (e.g. partial correctness proofs of programs ignoring the termination problems, type checking, program optimizations which are not carried in the absence of certainty about their feasibility, ...).

Abstract program properties are modeled by a complete semilattice, Birkhoff[61]. Elementary program constructs are locally interpreted by order preserving functions which are used to associate a system of recursive equations with a program. The program global properties are then defined as one of the extreme fixpoints of that system, Tarski[53]. 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 build a more abstract model of the semantics of programs, in that it ignores the sequencing of control flow. This model is taken to be the most concrete of the abstract interpretations of programs. Section 5 gives the formal definition of the abstract interpretations of a program.

3. Syntax and Semantics of Programs

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

3.1 Syntax of a Program

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

\[ n \text{-succ, } n \text{-pred : Nodes} \rightarrow 2\text{Nodes} \quad (m \in n \text{-succ}(n)) \]

\[ \iff (n \in n \text{-pred}(m)) \]

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

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

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

* Attaché de Recherche au C.N.R.S., Laboratoire Associé n° 7.
** This work was supported by IRIA-SESORI under grants 75-035 and 76-160.
3.2 Semantics of Programs

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

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

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

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

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

In particular the projection $\text{val} | \text{Bexpr}$ of the function $\text{val}$ in domain $\text{Bexpr}$ has the functionality:

- $\text{val} | \text{Bexpr} : \text{Bexpr} \rightarrow \text{Values}$

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

- $\text{States} = \text{Arcs}^2 \times \text{Env}$.

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

- $\forall s \in \text{States}, s = (cs(s), \text{env}(s))$

- We use a continuous conditional function $\text{cond}(b, e_1, e_2)$ to denote $\text{cond}(b, e_1, e_2)$.

- We use a continuous conditional function $\text{cond}(b, e_1, e_2)$ that is true, false or $\tau$ respectively as the value of $b$ is $\top$, $\bot$, false or $\tau$. We also use $\text{cond}(b, e_1, e_2)$.

- 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):

- $n\text{-state} : \text{States} \rightarrow \text{States}$

- $n\text{-state}(s) = (n_{\text{state}}(s))$ for all $s \in \text{States}$.

- Let $n$ be end$(cs(s))$, $e$ be $(\text{env}(s))$ within $\text{case}$ in $\text{Assignments}$

- $n_{\text{state}}(s) = \text{cond}(\text{val} (\text{test}(n)) (e), \text{Bexpr} (\text{a-succ}(n), e), (\text{a-succ-f}(n), e))$

- $n_{\text{state}}(s) = \text{Ends} (\text{success}(n), e)$

- $n_{\text{state}}(s) = s$

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

- Let $\bot_{\text{Env}}$ be the bottom function on $\text{Env}$ such that:

- $(\forall e \in \text{Ident}^0, \bot_{\text{Env}}(x) = \bot_{\text{Values}})$

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

- $I\text{-states} = \text{val}(\text{a-succ}(m), \bot_{\text{Env}})$ for all $m \in \text{Entries}$.
A "computation sequence" with initial state \( i_s \in I\text{-}states \) is the sequence:

\[
\sigma^n = \text{n-state}^n(i_s)
\]

for \( n = 0, 1, \ldots \),

where \( f^n \) is the identity function and \( f^0 = f \).

The initial to final state transition function:

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

is the minimal fixpoint of the functional:

\[
\lambda F. \ (\text{n-state} \circ F)
\]

Therefore

\[
\text{n-state}^\omega = \nu \text{States} \cdot \text{States}(\lambda F. \ (\text{n-state} \circ F))
\]

where \( \nu \text{States} \) denotes the least fixpoint of \( F : D \to D \), Tarski[55].

4. Static Semantics of Programs

The constructive or operational semantics of programs defined in section 3 considers the sequence in which states occur during execution. The fundamental remark of Floyd[67] is that to prove static properties of programs it is often sufficient to consider the sets of states associated with each program point.

Hence, we define the context \( Cq \) at some program point \( q \in \text{Arcs of a program} \) to be the set of all environments which may be associated to \( q \) in all the possible computation sequences of \( P \):

\[
Cq = \{ e | (\exists n \geq 0, i_s \in I\text{-}states | <q, e> = \text{n-state}^n(i_s)) \}
\]

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

\[
Cv = \lambda q. (e | (\exists n \geq 0, i_s \in I\text{-}states | <q, e> = \text{n-state}^n(i_s)))
\]

According to the semantics of programs, the context \( Cv(r) \) associated to arc \( r \) is related to the contexts \( Cv(q) \) at arcs \( q \) adjacent to \( r \).

\( (\text{end}(q) = \text{origin}(r), \xrightarrow{r} \text{to}) \). From the definition of the state transition function we can prove the equation:

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

where

\[
\text{n-context} : \text{Arcs}^0 \times \text{Context-Vectors} \to \text{Contexts}
\]

is defined by:

\[
\text{n-context}(r, Cv) = \begin{cases} 
\text{case} & \text{origin}(r) \text{ in Entries} = (\text{Env}) \\
& \text{Assignments} \cup \text{Tests} \cup \text{Junctions} = \bigcup \text{env-on}(r)(\text{n-state}(q, e)) \\
& q \in a\text{-pred}((\text{origin}(r))) \end{cases}
\]

(We define \( \text{env-on} : \text{Arcs}^0 \times \{\text{States} \times \text{Env}\} \) to be \( \lambda r. (\lambda s. \text{cond}(r = \text{cs}(s), \{\text{env}(s)\}, \emptyset)) \).

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

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

where

\[
\text{F-cont} : \text{Context-Vectors} \to \text{Context-Vectors}
\]

is defined by:

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

Context-Vectors is a complete lattice with the ordering \( \sqsubseteq \) such that:

\[
Cv_1 \sqsubseteq Cv_2 = \lambda r. (Cv_1(r) \sqsubseteq Cv_2(r)).
\]

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

\[
\{Cv \subseteq Cv_2\} \iff (\forall r \in \text{Arcs}, Cv(r) \subseteq Cv_2(r))
\]

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

\[
\text{D1} : Cv = \lambda q. (e | (\exists n \geq 0, i_s \in I\text{-}states | <q, e> = \text{n-state}^n(i_s)))
\]

or

\[
\text{D2} : Cv = \nu \text{Context-Vectors}(\text{F-cont})
\]

The concrete context vector \( Cv \) is such that for any program point \( q \in \text{Arcs of the program} \):

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

\[
\{ e \geq 0, i_s \in I\text{-}states | <q, e> = \text{n-state}^n(i_s) \} \implies (e \in Cv(q))
\]

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

\[
(\forall e \in Cv(q)) \implies \{ i_s \geq 0, i_s \in I\text{-}states | <q, e> = \text{n-state}^n(i_s) \}
\]

\( Cv \) is merely a static summary of the possible executions of the program. However, our definitions \( \text{D1} \) or \( \text{D2} \) of \( Cv \) cannot be utilized at compile time since the computation of \( Cv \) consists in fact in running the program (for all the possible input data). In practice compilers may consider states which can never occur during program execution (e.g., some compilers consider that any program may always perform a division by zero although this is not the case for most programs). Hence compilers may use "abstract" contexts satisfying \( (a) \) but not necessarily \( (b) \), which therefore correctly approximate the concrete contexts we considered until now.

5. Abstract Interpretation of Programs

5.1 Formal Definition

An abstract interpretation \( I \) of a program \( P \) is a tuple

\[
I = \langle A\text{-Cont}, \preceq, \leq, r, \lambda, \text{Int} \rangle
\]

where the set of abstract contexts is a complete semilattice with ordering \( \preceq \), \( \{ x \preceq y \} \iff \{ x \leq y \} \). This implies that \( A\text{-Cont} \) has a supremum \( r \). We suppose also \( A\text{-Cont} \) to have an infimum \( \lambda \).
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 = $A$-Cont$^2$ with $A$-Cont$^2$ may be, we define:

$$
Cv' \triangleq Cv'' = (\lambda \cdot Cv'(x) \cdot Cv''(y))
$$

Where $\bar{r} \in \text{Arcs}^0$, $Cv'(x) \leq Cv''(y)$

$$
\bar{r} = \lambda \cdot \bar{r} \cdot \bar{l}
$$

$A$-Cont, $\sim$, $\perp$, $\triangleright$, $\triangleright$ can be shown to be a complete lattice. The function $\text{Int} : \text{Arcs}^0 \times A$-Cont $\rightarrow$ A-Cont defines the interpretation of basic instructions. If $\{C(q) \mid q \in a-$pred$(n)\}$ is the set of input contexts of node $n$, then the output context on exit arc $r$ of $n$ ($r \in a-$succ$(n)$) is equal to $\text{Int}(r, C)$. Int is supposed to be order-preserving?

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

$$
Cv' \triangleq Cv'' \Rightarrow \{ \text{Int}(a, Cv') \leq \text{Int}(a, Cv'') \}
$$

The local interpretation of elementary program constructs which is defined by Int is used to associate a system of equations with the program. We define $\text{Int} : A$-Cont $\rightarrow$ A-Cont | $\text{Int}(Cv) = \lambda \cdot \text{Int}(r, Cv)$

It is easy to show that $\text{Int}$ is order-preserving. Hence it has fixpoints, 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 $Cv = \text{Int}(Cv)$.

5.2 Typology of Abstract Interpretations

The restriction that "A-Cont" must be a complete semi-lattice is not drastic since Mac Neille[37] showed that any partly ordered set $S$ can be embedded in a complete lattice so that inclusion is preserved, together with all greatest lower bounds and lowest upper bounds existing in $S$. Hence in practice the set of abstract contexts will be a lattice, which can be considered as a join ($\cup$) 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 $n$ or $\cup$ operator represents the effect of path converging. The real need for this operator is to define completeness which ensures Int to have extreme fixpoints (see 8.4).

The result of an abstract interpretation was defined as a solution to forward ($\rightarrow$) equations: the output contexts on exit arcs of node $n$ are defined as a function of the input contexts on entry arcs of node $n$. One can as well consider a system of backward ($\leftarrow$) equations: a context may be related to its successors. Both systems ($\rightarrow$, $\leftarrow$) may also be combined.

Finally we usually consider a maximal ($\triangleright$) or minimal ($\triangleright$) solution to the system of equations, (by agreement, maximal and minimal are related to the ordering $\leq$ defined by ($x \leq y \rightarrow (x \cup y = y)$

$\Rightarrow (x \cap y = x)$). However known examples such as Manna and Shanin[75] show that the suitable solution may be somewhere between the extreme ones.

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

\[ (u, \triangleright) \quad (n, \triangleright) \]
\[ (u, \triangleright) \quad (n, \triangleright) \]

Examples:

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

5.3 Examples

5.3.1 Static Semantics of Programs

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

$\text{Int} : \langle \text{Contexts}, \cup, \leq, \text{Env}, \emptyset, n-$context$\rangle$

where $\text{Contexts}$, $\cup$, $\leq$, $\text{Env}$, $\emptyset$, $n-$context, $\text{Context}$-Vectors, $\cup$, $\leq$, $F$-Context respectively correspond to $A$-Cont, $\sim$, $\perp$, $\triangleright$, $\triangleright$, $\text{Int}$, $A$-Cont, $\sim$, $\perp$, $\triangleright$.

5.3.2 Data Flow Analysis

Data flow analysis problems (see references in Ulman[75]) may be formalized as abstract interpretations of programs.

"Available expressions" give a classical example. An expression is available on 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:

$B$-vector : $\text{Expr}_P \times \{\text{true}, \text{false}\}$

$B$-vector is clearly a complete boolean lattice. The interpretation of basic nodes is defined by:

```
avail(r, By)
```

```
let n be origin(r) within case n in
  Entries => \lambda . \text{false}
  Assignments \cup \text{Tests} \cup \text{Junctions} =>
  \lambda e. (\text{generated}(n)(e) \lor (\text{and} \lor \text{By}(p)(e))
  \text{pred}(n)
  \text{and} \text{transparent}(n)(e))

\text{esac}
```

(Nothing is available on entry arcs. An expression $e$ is available on arc $r$ (exit of node $n$) if either the expression $e$ is generated by $n$ or for all predecessors $p$ of $n$, $e$ is available on $p$ and $n$ does not modify arguments of $e$).

The available expressions are determined by the maximal solution (for ordering $\lambda e. \text{false} \leftarrow \lambda e. \text{true}$) of the system of equations:

$\text{By} = \text{avail}(By)$
The determination of available expressions, back-
dominators, intervals, ... requires a forward sys-
tem of equations. Some global flow problems, nota-
ably the live variables and very busy expressions
require propagating information backward through
the program graph, they are examples of backward
systems of equations.

5.3 Remarks

Our formal definition of abstract interpretations
has the completeness property since the model en-
ures the existence of a particular solution to
the system of equations and therefore defines at
least some global property of the program. It must
also have the consistency property, that is 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 \( \bar{\mathcal{I}} = \langle \bar{\mathcal{I}}, \bar{\mathcal{I}}, \bar{\mathcal{I}}, \bar{\mathcal{I}}, \bar{\mathcal{I}} \rangle \) of a program is consistent with a "con-
text" interpretation \( \mathcal{I} = \langle \mathcal{I}, \mathcal{I}, \mathcal{I}, \mathcal{I}, \mathcal{I} \rangle \) if the context vector \( \bar{\mathcal{C}} \) resulting from \( \bar{\mathcal{I}} \) is a cor-
rect approximation of the context vector \( \mathcal{C} \) result-
ing from the more refined interpretation \( \mathcal{I} \). This
may be rigorously defined by establishing a corre-
pondence (\( \gamma : \text{abstraction} \)) between concret-
et context vectors, and inversely (\( \gamma : \text{concreti-
zation} \)), and requiring:

6.0 \( \langle \mathcal{C}, \gamma(\mathcal{C}) \rangle \) and \( \langle \bar{\mathcal{C}}, \gamma(\bar{\mathcal{C}}) \rangle \)

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

If \( \mathcal{I} : \bar{\mathcal{I}} \) we note \( \bar{\mathcal{D}} = \mathcal{D}_\mathcal{I} \) and \( \bar{\mathcal{D}}_\mathcal{I} = \mathcal{D}_\mathcal{I} \) and
\( \bar{\mathcal{F}} = \mathcal{F}_\mathcal{I} \) and \( \mathcal{F}_\mathcal{I} = \mathcal{F}_\mathcal{I} \).

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

6.1 \( \alpha : \mathcal{A} \rightarrow \bar{\mathcal{A}} \), \( \gamma : \mathcal{A} \rightarrow \mathcal{A} \)

6.2 \( \alpha \) and \( \gamma \) are order-preserving

6.3 \( \forall x \in \mathcal{A} \), \( x \rightarrow \gamma(x) \)

6.4 \( \forall x \in \mathcal{A} \), \( x \rightarrow \gamma(x) \)

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 intro-
duces the idea of approximation: the abstraction \( \alpha(\mathcal{C}) \) of
a concrete context \( \mathcal{C} \) may introduce some loss of
information so that when concretizing again \( \gamma(\alpha(\mathcal{C})) \)
we may get a larger context \( \gamma(\alpha(\mathcal{C})) \) \( \mathcal{C} \). 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:

\[
\begin{align*}
\Psi(a, x) &\in \mathcal{A} \times \mathcal{A}, \\
\gamma(\mathcal{A}(a, x)) &\geq \mathcal{A}(a, \gamma(x)) \\
6.5 &\quad \text{and} \\
\Psi(a, x) &\in \mathcal{A} \times \mathcal{A}, \\
\mathcal{A}(a, \gamma(x)) &\leq \mathcal{A}(a, \gamma(x))
\end{align*}
\]

These two hypothesis are in fact equivalent (lemma
L2 in appendix I2). The following schema illustrates
6.5, i.e. the idea of abstract simulation of
concrete computations:

\[
\begin{align*}
\mathcal{I} &\rightarrow \mathcal{C}_0 \\
\mathcal{I} &\rightarrow \mathcal{C}_1 \\
\mathcal{I} &\rightarrow \mathcal{C}_2
\end{align*}
\]

Suppose we want to compute the concrete output con-
text \( \mathcal{C}_0 \) (associated with arc \( a \)) resulting from con-
crete input contexts \( \mathcal{C}_1 : \mathcal{C}_0 = \mathcal{A}(a, \mathcal{C}_1) \). We can
as well approximate this computation in the abstract
universe, and get \( \mathcal{C}_0 = \gamma(\mathcal{A}(a, \mathcal{C}_1)) \). 6.5 requires
\( \mathcal{C}_0 \) to contain at least \( \mathcal{C}_0 \), that is \( \mathcal{C}_0 \leq \mathcal{C}_0 \). On the
contrary we do not require \( \mathcal{C}_0 \) to contain at most
\( \mathcal{C}_0 \), that is \( \mathcal{C}_0 \leq \mathcal{C}_0 \) is not compulsory.

We will say that \( I \) is a refinement of \( \bar{I} \), or that
\( I \) is an abstraction of \( \bar{I} \), denoted \( I \leq (\alpha, \gamma) \bar{I} \), if
and only if there exist \( \alpha \) and \( \gamma \) satisfying hypothe-
sis 6.1 to 6.5.

Note that \( I \leq (\alpha, \gamma) \bar{I} \) imposes a local consistency
of the interpretations \( \mathcal{I} \) and \( \bar{I} \), at the level of pr-
imitive language construcs (6.5). Theorems T1 and
T2 of Appendix 12 then prove 6.0 which defines the
global consistency of \( \mathcal{I} \) and \( \bar{I} \) at the program level.

In particular if we take

\( \mathcal{I} \) any abstract interpretation \( \bar{I} \) of \( P \), consistent with
\( \mathcal{I} \) is consistent with the seman-
tics of \( P \), which implies:

\[
\forall x \in \mathcal{A} \text{, let } \gamma(x) \text{ be the result of } \bar{I}, \\
\text{if } \langle \mathcal{C}, \gamma(x) \rangle \text{ and } \mathcal{C} \text{, then } \mathcal{C} \subseteq \mathcal{C}.
\]

As previously noticed, the abstract interpretation
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 Pred over the program variables \( (x_1, \ldots, x_n) \) which
are the free variables in the predicate. The
abstract interpretation is then:

\[
\mathcal{I} \rightarrow \langle \text{Pred, or, } \Rightarrow, \text{ true, false, n-pred} \rangle
\]
where $\text{n-pred}$ defines Floyd's strongest post condition:

\[
\text{n-pred}(r, Pv) = \begin{cases} \text{let}(n \text{ be } \text{origin}(r), (p \text{ be } \text{a-pred(origin}(r)) \text{ wthin} \text{ case } r \text{ in} \text{ Entries} \iff (\exists x \in \text{Ident}, x = 1 \text{ Values}) \end{cases} \]

\[
\text{or} \quad q = \text{a-pred}(n) \quad (Pv(q))
\]

\[
\text{Tests} \iff \begin{cases} \text{a-succ-t}(n) \iff Pv(p) \text{ and } \text{test}(n) \end{cases} \]

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

\[
\text{esac}
\]

\[
\text{Assignments} \iff \begin{cases} \text{let}(p \text{ be } \text{Pv}(p),(x \text{ be } \text{id}(n)), (e \text{ be } \text{expr}(n)) \text{ wthin} \text{ (lv } e\text{Values} | P[v/x] \text{ and } x = e(v/x)) \end{cases} \]

\[
\text{esac}
\]

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

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

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

\[
e \in C \quad \text{x } \in \text{Ident}
\]

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

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

\[
\forall a \in \text{ArCs}, \forall P \in \text{Pred}, \exists a(\equiv \text{context}(a, \gamma(Pv))) \Rightarrow \text{n-pred}(a, P) \}
\]

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

\[
\forall q \in \text{ArCs}, \forall P \text{ be the result of } I_{GS}'
\]

\[
\begin{cases} \text{ln } 0 \text{, } l \in S \text{ states} | q, e = \text{n-state}^n(l, q) \end{cases} \iff (Pv(q) \Rightarrow \alpha(e))
\]

7. The Lattice of Abstract Interpretations

The relation $\leq$ comparing the levels of abstraction of two interpretations is a quasi-ordering since it is:

- reflexive: $I \leq (I, 1)I$ where $1 = \lambda x. x$ is the identity function,

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

The relation $\equiv$ on abstract interpretations defined by:

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

is an equivalence relation. We have:

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

The proof gives some insight in the abstraction process:

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

2 - reciprocally,

If $I \leq (\alpha_1, \gamma_1)I'$, let $\equiv (\alpha, \gamma)$ be the equivalence relation defined on $I$ (properly speaking, on the set of abstract contexts of I) by:

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

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

\[
\alpha_1 | \gamma_1(I') \quad \text{and} \quad \alpha_2 | \gamma_2(I) \quad \text{are bijections, hence}
\]

\[
\forall x' < I', \exists x' | \gamma_1(I') \text{ such that:}
\]

\[
x' = (\alpha_1 | \gamma_1(I'))(x). \text{ Likewise, } x' \in \gamma_1(I') \Rightarrow x' = (\alpha_2 | \gamma_2(I))(x'').
\]

Therefore, $\forall x' < I', \exists x'' | \gamma_2(I') \text{ such that:}

\[
x' = (\alpha_1 | \gamma_1(I')) \circ (\alpha_2 | \gamma_2(I))(x'').
\]

Thus

\[
(\alpha_1 | \gamma_1(I')) \circ (\alpha_2 | \gamma_2(I)) \text{ is a bijection between } \gamma_2(I) \text{ and } I'.
\]

Since $(\alpha_2 | \gamma_2(I))^{-1} \text{ is a bijection between } I \text{ and } \gamma_2(I)$, the composition

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

is a bijection between $I$ and $I'$, hence $\alpha_1$ is a bijection between $I$ and $I'$ which is trivially an algebraic morphism. ($\alpha_1$ is isotone, its inverse $\alpha_1^{-1} = \gamma_1$ is isotone and $\alpha_1(\text{Int}(a, X)) = \text{Int}'(a, \gamma_1(X))$ Q.E.D.

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

In particular, we can restrict $I$ to be set of interpretations which abstract $I_{GS}$. $I$ is then a lattice, (with ordering $\leq$) 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 $\alpha(S) = [\min(S), \max(S)]$. Then $S$ is finite the bounds will eventually be $-\text{or } +\text{.}

$\gamma((a, b)) = (x | a \leq x \leq b)$. The abstract contexts are then, (Cousot[76]):

243
8. Abstract Evaluation of Programs

The system of equations:

\[ \text{CV} : \overline{\text{Int}}(\text{CV}) \]

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

\[
\text{CV} : (C := I; \text{ until } C = \overline{\text{Int}}(C) \text{ do } C := \overline{\text{Int}}(C) \text{ repeat } C)
\]

8.1 Correctness

If \( \text{Int} \) is supposed to be a complete morphism (i.e. infinitely distributive over \( \cdot \)) then \( \text{CV} \) is the least fixpoint of \( \overline{\text{Int}} \). (e.g. Kildall[73], 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 \( CV \) of Kleene's sequence can also be shown to be the least fixpoint of \( \overline{\text{Int}} \) (e.g. Wegbreit[75], since in a well-founded semi-lattice, any isotine function is continuous). Finally, if \( \text{Int} \) is only supposed to be isotine, \( \text{CV} \) is an approximation \((\varepsilon)\) of the least fixpoint (e.g. 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. 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 (Stutzer[75]) or heuristics (Katz and Nmhasi[76]) may be used to pass to the limit, or approximate it, (Cousot[76]).

8.3 Efficiency

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

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

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

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

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

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

The main difficulty is to obtain the probability \( \text{Prob}(\text{test}(n) = \text{true}) \) of taking the true path at a test node \( n \). Suppose the values of these probabilities can be determined (from hypothesis on the input data).

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

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

then

\[
\max_{1 \leq p \leq n} C_v(p) = \max_{1 \leq p \leq n} C_v(p) = \max_{1 \leq p \leq n} C_v(p)
\]

and

\[
\max_{1 \leq p \leq n} (n_1 \ast q) = \max_{1 \leq p \leq n} (n_1 \ast q).
\]

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

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

\[
\text{- Let } P \text{ be the program "begin while } T \text{ do } I \text{ end".} \\
\text{The number } n \text{ of times the expression } T \text{ is tested is given by the minimal solution to the equation } n = 1 + \frac{q}{1 - q} \text{ where } q \text{ is the probability of } T \text{ to be true. } n \text{ may be determined by the limit of Kleene's sequence:} \\
\text{0 + 1 + } q + q^2 + \ldots + q^\infty \text{ which is an infinite series. Its sum is } \frac{1}{1 - q}. \\
\]

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 \( \tilde{I} \) (in \( I \)) may be used for that purpose (e.g., Tenenbaum[74]). It is often better to make approximations in \( I \), for example by "accelerating the convergence" of Kleene's sequences.

9.1 Finite Iterative and Increasing Approximation of the Least Fixpoint Starting from a Lower Bound

Let \( I = (A-\text{Cont}, \ast, \cdot, \leq, l, \ast, \text{Int}) \) be an interpretation of \( P \). When the least fixpoint \( C_v \) of \( \text{Int} \) is unreachable, we look for an upper bound \( U_b \) of \( C_v \), since according to the correctness requirement \( S_0 \leq C_v \leq \gamma(C_v) \).

9.1.1 Increasing Approximation Sequence

Let \( \hat{\text{Int}} = \text{Cont} \ast \text{Int} \ast \text{Cont} \ast \ldots \)

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

\[
\Rightarrow (C \leq \text{Int}(C) \leq \hat{\text{Int}}(C)).
\]

9.1.1.2 Every infinite sequence \( \hat{\text{Int}}, \hat{\text{Int}}^2(\hat{\text{Int}}), \ldots \)

\( \hat{\text{Int}}^n(\hat{\text{Int}}), \ldots \) is not strictly increasing.

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

9.1.1.3 \( S_0 = \hat{I} \)

\[
S_{n+1} = \begin{cases} 
\text{if } \text{not}(\text{Int}(S_n) \leq S_n) & \text{then } \\
\hat{\text{Int}}(S_n) & \\
\text{else } S_n 
\end{cases}
\]

We now prove that \( S_n \) finite such that :

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

Let \( m \) be the least natural number (eventually infinite) such that \( S_m = S_{m+1} \). \( \forall k \in \mathbb{N}, m \leq k \), we know from 9.1.1.3 that \( \text{not}(\text{Int}(S_k) \leq S_k) \). Whence by definition of the ordering, \( S_k \neq \text{Int}(S_k) = S_k \).

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

\[
S_{k+1} = \hat{\text{Int}}(S_k) = \text{Int}(S_k) = S_k
\]

and therefore we conclude \( S_{k+1} = S_k \). \( \forall k \in \mathbb{N}, m \leq k \). 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 \( X \in \text{Cont} \) such that \( \text{Int}(X) \not\subseteq X \) (Tarski[55]) hence :

\[
\forall X \in \text{Cont}, (\text{Int}(X) \not\subseteq X) \Rightarrow (C_v \not\subseteq X)
\]

Since \( S_m = S_{m+1} \) we have \( \text{Int}(S_m) \not\subseteq S_m \) and therefore \( C_v \not\subseteq S_m \). \( S_m \) is a correct approximation of \( C_v \).
When $A$-Cont satisfies the ascending chain condition, one can choose $A$-Int to be Int and therefore the approximation sequence generalizes Kleene’s sequence and the related methods.

### 4.1.2 Meaning 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 $A$-Int by local modifications to Int.

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

- $(q, C) \in \text{Arcs} 
  \Rightarrow
  \text{Int}(q, C) \neq \text{Int}(q, C / r))$

(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 $q_1, \ldots, q_n$ of arcs, such that $q_i \in \{1, m, \text{Cv}(q_{i+1})\}$ depends on $Cv(q_i)$ and $Cv(q_i)$ depends on $Cv(q_i)$, e.g. in a forward interpretation a cycle corresponds to a loop in the program.

In any infinite strictly increasing Kleene’s sequence $Cv_0, \ldots, C$, ... since Arcs is finite there is some arc $q$ for which the sequence $Cv(q), \ldots$, $Cv(q)$, ... never stabilizes. Therefore $q$ must belong to a cycle or the contexts associated to $q$ transitively depend on the contexts associated to some other arc $r$ which itself belongs to a cycle. The sequence of contexts associated to any arc of that cycle never stabilizes. In order to avoid this phenomenon, we introduce:

- The binary operation $\vee$ called widening defined by:
  \begin{align*}
  & 9.1.3.1 \forall : A\text{-Cont} \times A\text{-Cont} \rightarrow A\text{-Cont} \\
  & 9.1.3.2 \forall(C, C') \in A\text{-Cont} \Rightarrow C \vee C' \subseteq C' \\
  & 9.1.3.3 \forall A\text{-int sequence } S_0, \ldots, S_n \ldots \\
  & \text{of the form } S_0 = C_0, \ldots, S_n = C_n \vee C_n, \ldots \\
  & \text{(where } C_0, \ldots, C_n \text{ are arbitrary abstract contexts) is not strictly increasing.}
  \end{align*}

- 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: $\forall q \in W$-arcs. (e.g. in a forward interpretation on a reducible program graph, $W$-arcs may be chosen to be the set of exit arcs of the junction nodes which are interval headers. On irreducible graphs an arbitrary choice has to be made so that any loop of the program goes through a widening arc).

- The approximate interpretation $A$-Int : Arcs $\times$ A-Cont $\rightarrow$ A-Cont defined by:
  \begin{align*}
  & 9.1.3.4 A\text{-int} = \lambda(q, Cv). \begin{cases}
  & \text{if } q \in W\text{-arcs then} \\
  & Cv(q) \vee \text{Int}(q, Cv)
  \end{cases}
  \\
  & \text{else} \\
  & \text{Int}(q, Cv)
  \end{align*}

As before, we define:

\begin{align*}
  9.1.3.5 A\text{-int} = \lambda(Cv, (\lambda q. A\text{-int}(q, Cv)))
\end{align*}

Now we have to show that this definition of $A$-Int satisfies the requirements 9.1.1.2 and 9.1.1.1.

Let us consider a sequence $S_0 = 1, \ldots, S_{n+1}$, ... We show that this sequence is increasing that is to say:

\begin{align*}
  9.1.3.6 S_{n} \leq A\text{-int}(S_n), \forall n \geq 0.
\end{align*}

Trivially for $n = 0$, $S_0 = 1 \leq A\text{-int}(S_0)$. For the induction step, suppose the result to be true for $n \leq m$. Let us prove that:

\begin{align*}
  S_{m+1} \leq A\text{-int}(S_{m+1}) \\
  \Rightarrow \quad S_{m+1}(q) \leq A\text{-int}(q, S_{m+1}), \forall q \in \text{Arcs}.
\end{align*}

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

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

\begin{align*}
  \Rightarrow \quad \text{Int}(q, S_{m}) \leq A\text{-int}(q, S_{m+1}) & \text{ since } S_{m} \subseteq S_{m+1} \text{ and Int is order preserving. Moreover from 9.1.3.4 we get Int}(q, S_{m}) = A\text{-int}(q, S_{m}) \text{ and therefore } S_{m+1}(q) \leq A\text{-int}(q, S_{m+1}).
\end{align*}

Finally $S_{m+1} \leq A\text{-int}(S_{m+1})$, Q.E.D.

An infinite sequence $S_0 = 1, \ldots, S_n = A\text{-int}^n(1)$, cannot be strictly increasing since otherwise there would exist some widening arc for which the sequence $S_n(q), \ldots, S(q), \ldots$ would never stabilize thus contradicting 9.1.3.3.

We now prove 9.1.1.1 that is to say that:

\begin{align*}
  \forall n \geq 0, S_n = A\text{-int}^n(1)
\end{align*}

implies

\begin{align*}
  S_n = \text{Int}(S_n) \leq A\text{-int}(S_n)
\end{align*}

\begin{align*}
  \Rightarrow (S_n = \text{Int}(S_n))(q) \leq A\text{-int}(S_n)(q), \forall q \in \text{Arcs}
\end{align*}

\begin{align*}
  \Rightarrow S_n(q) \leq \text{Int}(q, S_n) \leq A\text{-int}(q, S_n) \text{ (see 9.1.3.5)}
\end{align*}

If $q \in W$-arcs, we have $A\text{-int}(q, S_n) = S_{n}(q) \vee \text{Int}(q, S_{n}) = S_{n}(q) = A\text{-int}(q, S_{n})$ by 9.1.3.2. If now $q \notin W$-arcs we must show:

\begin{align*}
  S_{n}(q) \leq \text{Int}(q, S_{n}) \leq \text{Int}(q, S_{n})
\end{align*}

\begin{align*}
  \Rightarrow S_{n}(q) \leq \text{Int}(q, S_{n}) \leq \text{Int}(q, S_{n})
\end{align*}

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

9.2 Example: Bounds of Integer Variables

In a PASCAL program operating on arrays, the compiler should ensure that arrays are subscripted only by indices within bounds. Furthermore one can use the lattice $\mathcal{L}$ of section 7. Let us take an obvious example:
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. \(C_1 = [1, 1]\)
3. \(C_2 = C_1 \cup C_4\)
4. \(C_3 = C_2 \cap [-\infty, 100]\)
5. \(C_4 = C_3 + [1, 1]\)
6. \(C_5 = C_2 \cap [101, +\infty]\)

Assignment statements are treated using an interval arithmetic (e.g. \([i, j] + [k, l] = [i+k, j+k]\) 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. \([\_\_] < [0, 0] < [0, 1] < \ldots < [0, +\infty]\) for the program \(x := 0\); while \(\text{true}\) do \(x := x + 1\); we must use an approximation sequence. Hence the results will be somewhat inaccurate but runtime subcost tests may be inserted in the absence of certainty.

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

\[-i, j\] \(\triangledown\) \([k, l] = \begin{cases} [i, \min(j, k)] & \text{if } k < i \text{ then } \Rightarrow \text{ else } i \{i, l]\end{cases}\]

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

\(C_2 = C_2 \triangledown (C_1 \cup C_4)\)

The corresponding approximation sequence is:

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

The 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.

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

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

\(S_0 \geq \tilde{\text{Int}}(S_0) \geq \tilde{\text{Int}}^2(S_0) \geq \tilde{\text{Int}}^n(S) \geq \tilde{\text{Int}}^n(S)\)

If \(S_n\) is not a fixpoint of \(\tilde{\text{Int}}\) and the above descending sequence is finite (e.g. the lattice \(A\text{-Int}\) is a fixpoint, its limit is a better approximation of \(C\) than \(S_n\). When the sequence is infinite or slowly converging, one can among other solutions approximate its limit.

### 9.3.1 Decreasing Approximation Sequence

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

\(\tilde{\text{Int}}^n(S_n) \geq \tilde{\text{Int}}^n(S_0)\)

In order to accelerate the convergence, we start for the next step find an approximation \(D\) such that:

\(\tilde{\text{Int}}^n(S_n) \geq D \geq \tilde{\text{Int}}(D)\)

But not knowing \(C\), this characterization is very weak since \(D\) could be incomparable with \(C\). The fact that \(C\) is the greatest lower bound of the set of \(X \in A\text{-Int}\) such that \(\tilde{\text{Int}}(X) \geq X\) gives a correctness criterion for a choice of \(D\) when \(C\) is unknown. We must have:

\(\tilde{\text{Int}}^{n+2}(S) \geq D \geq \tilde{\text{Int}}(D)\)

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

### 9.3.2 Truncated Decreasing Sequence

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

\(0 \geq D \geq \tilde{\text{Int}}^{n+2}(S)\)

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 \( \forall C \in A_{\text{Cont}} \)
\( \{ C \supseteq \text{Int}(C) \} \Rightarrow \{ C \supseteq D_{\text{Int}}(C) \supseteq \text{Int}(C) \} \)

9.3.2.2 \( \forall C \in A_{\text{Cont}}, \) every infinite sequence \( C, D_{\text{Int}}(C), \ldots, D_{\text{Int}}(C), \ldots \) is not strictly decreasing.

The truncated decreasing sequence \( s_0', \ldots, s_n' \) is recursively defined by:

9.3.2.3 \( s_0' = s_m \)
\( s_{n+1}' = \begin{cases} s_n' & \text{if } (s_n' \neq \text{Int}(s_n')) \\
\text{else} & \text{then } D_{\text{Int}}(s_n') \end{cases} \)

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:

\( s_0' = s_m \supseteq \text{Int}(s_0') \supseteq CV \)

If \( p > 0 \) then \( s_p' \neq \text{Int}(s_p') \), therefore \( s_0' \supseteq \text{Int}(s_0') \).

Then applying 9.3.2.1 we have:

\( s_0' \supseteq D_{\text{Int}}(s_0') = s_1' \supseteq \text{Int}(s_1') \supseteq CV \)

But 9.3.2.3 implies \( s_0' \neq D_{\text{Int}}(s_0') \), hence:

\( s_0' \supseteq s_1' \supseteq \text{Int}(s_0') \supseteq CV \)

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

\( s_k' \supseteq \text{Int}(s_k') \supseteq CV \)

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

\( \text{Int}(s_k') \supseteq \text{Int}(s_{k-1}') \supseteq \text{Int}(CV) \)

By transitivity \( s_k' \supseteq \text{Int}(s_k') \) and since 9.3.2.3 implies \( s_k' \neq \text{Int}(s_k') \) we have from 9.3.2.1:

\( s_k' \supseteq D_{\text{Int}}(s_k') = s_{k+1}' \supseteq \text{Int}(s_k') \)

Since 9.3.2.3 implies \( s_k' \neq D_{\text{Int}}(s_k') \) we have:

\( s_k' \supseteq s_{k+1}' \supseteq \text{Int}(s_k') \supseteq CV \)

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 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 \( s_0 = \text{Int}P(s_0) \) is a fixpoint greater or equal to the least fixpoint \( CV \) of \( \text{Int} \).

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

9.3.4.2 \( \forall (C, C') \in A_{\text{Cont}}^2 \)
\( \{ C \supseteq C', C \supseteq C' \} \Rightarrow \{ C \supseteq C \Delta C' \supseteq C' \} \)

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

The approximated interpretation
\( D_{\text{Int}} : A_{\text{Cont}} \times A_{\text{Cont}} \rightarrow A_{\text{Cont}} \) is defined by:

9.3.4.4 \( D_{\text{Int}} = \lambda(q, CV) \cdot \begin{cases} CV(q) \Delta \text{Int}(q, CV) & \text{if } CV(q) \subseteq \text{Int}(q, CV) \\
\text{else } \text{Int}(q, CV) & \end{cases} \)

This definition of \( D_{\text{Int}} \) trivially satisfies the requirement 9.3.2.1 since \( CV \in A_{\text{Cont}} \) with property \( CV \supseteq \text{Int}(CV) \) implies \( CV(q) \supseteq \text{Int}(q, CV) \).

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) \( C_3 = C_2 \cap [\infty, 100] \)
(4) \( C_4 = C_3 + [1, 1] \)
(5) \( C_5 = C_3 \cap [101, +\infty] \)

The ascending approximation sequence led to the approximate solution:

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

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

\( \Delta[i, j] [k, \ell] = \begin{cases} [i, j] \cap [k, \ell] = [\text{if } i = -\infty \text{ then } k \text{ else } \min(i, k)] & \text{if } j = +\infty \text{ then } \ell \text{ else } \max(j, \ell) \end{cases} \)
Thus narrowing just discards infinite bounds and makes no improvement on finite bounds, it satisfies the requirements of 9.3.4. According to 9.3.4.4 the system of equations is modified by:

\[ C_2 = C_2 \Delta (C_1 \cup C_4) \]

The descending approximation sequence is:

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

\[ C_3 = [1, 101] \]

\[ C_4 = C_2 \cap [\neg \infty, 100] \]

\[ C_5 = [1, 101] \cap [101, +\infty] = [101, 101] \]

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 \( \sim \text{Cont} \) may be partitioned as follows:

\[ \text{\textup{\$fp}} \text{ and } \text{\textup{\$gfp}} \text{ are the least and greatest fixpoints of } \text{\textup{\textup{Int}}} \text{. The ascending (AKS) and descending (DKS) Kleene's sequences converge toward \textup{\textup{\$fp}} and \textup{\textup{\$gfp}} respectively. These limits are reached when } \text{\textup{\textup{Int}}} \text{ is continuous. When AKS is infinite we have proposed to use an ascending approximation sequence (AAS) to approximate \text{\textup{\textup{\$fp}}. Its limit may be some fixpoint \text{\textup{\textup{\textup{\$fp}}}, or some } S_m \text{ such that } S_m \sim \text{\textup{\textup{\textup{\textup{Int}}}(S_m)}} \text{ and } S_m \sim \text{\textup{\textup{\textup{\textup{\$fp}}}}.} \]
Any of the AAS, TDS, DAS, TAS methods may yields a fixpoint $fp$ which is not the fixpoint $\mathbf{fp}$ or $\mathbf{gfp}$ of interest. None of these methods can improve $fp$ to reach $\mathbf{fp}$ or $\mathbf{gfp}$, therefore a "fixpoint improvement method" is necessary. It is our feeling that such a method could be designed only when considering that A-Cont possesses a richer structure (i.e., for particular applications).

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

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

10. Conclusion

It is our feeling that most program analysis techniques may be understood as abstract interpretations of programs. Let us point out global data flow analysis in optimizing compilers [Kildall[73], Moore[1951], Hoare[1959, 1961], Ullman[1975], Wegbreit[1975], \ldots], type verification (Naur[1975], \ldots), type discovery (Cousot[1976], Sintzoff[1974], Tenenbaum[1974], \ldots), program testing (Henderson[1975], \ldots) symbolic evaluation of programs (Hewitt et al.[1973], Karr[1976], \ldots), program performance analysis (Wegbreit[1976], \ldots), formalization of program semantics (Hoare and Lauer[1974], Ligler[1975], Manna and Shamir[1975], \ldots), verification of program correctness (Floyd[1967], ParK[1969], Sintzoff[1975], \ldots), discovery of inductive invariants (Katz and Manna[1976], \ldots), proofs of program termination (Sist[1974], \ldots), program transformation (Sintzoff[1976], \ldots),

There is a fundamental unity between all these apparently unrelated program analysis techniques: a new interpretation is given to the program text which allows to build an often implicit system of equations. The problem is either to verify that a solution provided by the user is correct, or to discover or approximate such a solution.

The mathematical model we studied in this paper is certainly the weakest which is necessary to unify these techniques, and therefore should be of very general scope. It can be considerably enriched for particular applications so that more powerful results may be obtained.

Acknowledgements

We wish to thank M. Sintzoff for stimulating discussions. We were very lucky to have F. Blanc do the typing for us.

11. References


Cousot[1976]. Static determination of dynamic properties of generalized type unions. Submitted for publication. (Sept.)


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


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


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


12. Appendix

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

Note: L is a complete lattice.

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

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

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

<==>

{∀(x, y) ∈ L², (f(x) u y) ≥ f(x) u f(y))

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

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

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

g = u(x | (x ∈ L) ∧ (x ≤ F(x)))

τ = n(x | (x ∈ L) ∧ (F(x) ≥ x))

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

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

(H2.1): Let α and β be such that:

α: L → τ

γ: L → τ

α is order preserving

γ is order preserving

(H2.2): Let γ(x) < α(x)

(H2.3): Let γ(x) ≥ α(x)

γ(γ(x)) ≤ α(γ(x))

γ(α(x)) ≥ γ(x)

(H3.1): (H1), (H2), and {∀x ∈ L, F(α(x)) ≠ α(F(γ(x)))})

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

(L2): [H3.1] <== [H3.2]

Proof:

∀x ∈ L,

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

γ(F(x)) ≥ α(F(γ(x))) from H2.5

γ(F(x)) ≥ γ(α(γ(γ(x)))) from H2.4

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

γ(γ(α(x))) ≥ F(α(x)) in H3.2

γ(α(x)) ≥ x H2.6

F(γ(α(x))) ≥ F(x) F order preserving

γ(F(α(x))) ≥ F(x) transitivity

H2.3

α(γ(F(α(x)))) ≥ α(F(α(x))) H2.5

Q.E.D.

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

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

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

∀x ∈ L, {g u F(x) ≥ x} <== {g ≥ x}

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

∀x ∈ L, {l u F(x) ≤ x} <== {l ≤ x}
(T1): H1, H1', H2, H3 imply that the greatest fix-points g and \(\overline{g}\) of F and \(\overline{F}\) are related by:
\[
\{\alpha(g) \preceq \overline{g}\} \text{ and } (g \preceq \gamma(\overline{g}))
\]

Proof:

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

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

Q.E.D.

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

(T2): H1, H1', H2, H3 imply that the least fixpoints \(\underline{f}\) and \(\overline{f}\) of F and \(\overline{F}\) are related by:
\[
\{\gamma(\overline{f}) \preceq \underline{f}\} \text{ and } (\overline{f} \preceq \alpha(\underline{f}))
\]

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

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

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

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

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

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