Conference Record
of the
SIXTH ANNUAL ACM SYMPOSIUM ON
PRINCIPLES OF PROGRAMMING LANGUAGES

Papers Presented at the Symposium
San Antonio, Texas
January 29-31, 1979

Sponsored by the
ASSOCIATION FOR COMPUTING MACHINERY
1. INTRODUCTION and SUMMARY

Semantic analysis of programs is essential in optimizing compilers and program verification systems. It encompasses data flow analysis, data type determination, generation of approximate invariant assertions, etc.

Several recent papers (among others Cousot & Cousot[77a], Graham & Wegman[76], Kam & Ullman[76], Kildal[73], Rosen[76], Tarjan[76], Wegbreit[75]) have introduced abstract approaches to program analysis which are tantamount to the use of a program analysis framework \((A,t,Y)\) where \(A\) is a lattice of (approximate) assertions, \(t\) is an (approximate) predicate transformer and \(Y\) is an often implicit function specifying the meaning of the elements of \(A\). This paper is devoted to the systematic and correct design of program analysis frameworks with respect to a formal semantics.

In Section 6 we study and exemplify various methods which can be used in order to define a space of approximate assertions or equivalently an approximation function. They include the characterization of the least Moore family containing an arbitrary set of assertions, the construction of the least closure operator greater than or equal to an arbitrary approximation function, the definition of closure operators by composition, the definition of a space of approximate assertions by means of a complete join congruence relation or by means of a family of principal ideals.

Section 7 is dedicated to the design of the approximate predicate transformer induced by a space of approximate assertions. First we look for a reasonable definition of the correctness of approximate predicate transformers and show that a local correctness condition can be given which has to be verified for every type of elementary statement. This local correctness condition ensures that the (merge over all paths or fixpoint) global analysis of any program is correct since detour is not required for an execution.
shown that the space of approximate assertions can always be refined so that the merge over all paths analysis of a program can be defined by means of a least fixpoint of isotone equations.

Section 10 is devoted to the combination of program analysis frameworks. We study and exemplify how to perform the "sum", "product" and "power" of program analysis frameworks. It is shown that combined analyses lead to more accurate information than the conjunction of the corresponding separate analyses but this can only be achieved by a new design of the approximate predicate transformer induced by the combined program analysis frameworks.

2. PRELIMINARY DEFINITIONS

A program \( \pi \) is a pair \((V, G)\) where \( G \) is a program graph and \( V \) is the universe in which the program variables take their values.

The set \( L \) of elementary commands consists in elementary tests and elementary assignments: \( L = L_{\text{t}} \cup L_{\text{a}} \). An elementary test \( \alpha \in L_{\text{t}} \) is a total map from \( \text{dom}(\alpha) \subseteq V \) into \( B = \{\text{true}, \text{false}\} \). An elementary assignment \( \epsilon \in L_{\text{a}} \) is a total map from \( \text{dom}(\epsilon) \subseteq V \) into \( V \).

A program graph \( G \) is a tuple \((n_{\text{E}}, n_{\text{i}}, n_{\text{o}}, C)\) where \( n_{\text{E}} \) is the number of vertices (therefore \( n_{\text{E}} \geq 1 \)), \( E \subseteq \{1, n\}^{2} \) is the (non-empty) set of edges, \( n_{\text{i}} \in \{1, n\} \) is the entry point, \( n_{\text{o}} \in \{1, n\} \) is the exit point and \( C \subseteq E \cup L \) defines the command \( C(1, j) \) associated with each \( 1, j \) in \( E \). Let \( \text{pred}(\{1, n\}) = 2^{\{1, n\}} \) be \( \lambda j.\{j \in \{1, n\} : 1 \neq j \in E \} \) and \( \text{succ}(\{1, n\}) = 2^{\{1, n\}} \) be \( \lambda j.\{j \in \{1, n\} : j \neq 1 \in E \} \); then we assume that \( \text{pred}(\{1, n\}) = \emptyset, \text{succ}(\{1, n\}) = \emptyset \) and for any \( v \in \{1, n\} - \{1, n_{\text{i}}, n_{\text{o}}\} \), \( \text{pred}(v) \neq \emptyset \) and \( \text{succ}(v) \neq \emptyset \).

Example 2.0.1

The program:

\[
\begin{align*}
(1) & \quad \text{begin} \\
(2) & \quad \text{while } x \leq 100 \text{ do } \{ x \text{ is an integer variable} \\
(3) & \quad x := x+1; \quad \{ \text{no overflow can occur} \\
(4) & \quad \text{end}.
\end{align*}
\]

will be represented by its program graph:

\[
\begin{align*}
\lambda x. (x \leq 100) & \quad \lambda x. (x > 100) \\
\lambda x. (x \leq 100) & \quad \lambda x. (x > 100)
\end{align*}
\]

End of example.

If \( \text{Alg}(L, T, \text{all}, \text{all}) \) is a complete lattice, \( t \in L = (A \rightarrow A) \) and \( \phi \in A \), then the merge over all paths analysis of \( t \) using \( (A, \text{t}) \) and \( \phi \) \((\text{Alg}(L, T, \text{all}, \text{all})) \) is \( P \in A \) defined as:

\[
\forall t \in \text{Alg}(L, T, \text{all}, \text{all}), P = \bigcup_{P \in \text{path}(L)} \tilde{t}(P)(\phi)
\]

where \( \text{path}(L) \) is the set of paths from the point \( n_{i} \) to the vertex \( 1 \) and \( \tilde{t} : (E \rightarrow (A \rightarrow A)) \) is recursively defined as follows: if \( p \) is an empty path then \( \tilde{t}(p) \) is the identity map on \( A \) else \( p = (a, e) \) where \( q \in E, a \in E \) and \( \tilde{t}(p) = a \cdot \tilde{t}(a)(\tilde{t}(q)(\phi)) \).

The system of equations \( P = F_{\pi}(t, \phi)(P) \) associated with the program \( \pi \) using \((A, \text{t})\) and \( \phi \) is defined as follows:

\[
\begin{align*}
P_{1} & = \phi \\
P_{j} & = \bigcup_{t(C(1, j))} (P_{1}) \text{ if } j \in \{1, n\} - \{n_{i}\} - \text{pred}(j)
\end{align*}
\]

Notation: If \( \text{Alg}(L, T, \text{all}, \text{all}) \) is a complete lattice, then the set \( (L \rightarrow M) \) of total maps from the set \( L \) into \( M \) is a complete lattice \((L \rightarrow M) = \text{Alg}(L, T, \text{all}, \text{all}) \) for the pointwise ordering \( f \leq g \text{ if } \forall x \in L, f(x) \leq g(x) \).

In the following the distinction between \( E, L, T, \text{all}, \text{all} \) and \( E', L', T', \text{all}, \text{all} \) will be determined by the context.

Also a map \( f \in (L \rightarrow M) \) will be extended to \( (2^{L} \rightarrow 2^{M}) \) as \( \lambda x \in 2^{E}, f(x) : x \in S \) and to \((L \rightarrow M)\) as \( \lambda x \in 2^{L}, f(x) : x \in S \).

3. DEDUCTIVE SEMANTICS OF PROGRAMS

3.1 Forward Semantics

The forward semantic analysis of a program \( \pi \) consists in determining at each program point an invariant assertion which characterizes the set of states which are the descendants of the input states satisfying a given entry assertion \( \phi \).

More precisely an assertion is a total map from \( V \) into \( R \). The set \( \text{Alg}(L, T, \text{all}, \text{all}) \) of assertions is a complete boolean lattice partially ordered by the implication \( \rightarrow \).

Let \( \text{sp}(S)(P) \) be Floyd\'s strongest post-condition derived from the pre-condition \( P \in \text{Alg}(L, T, \text{all}, \text{all}) \) for the elementary command \( \text{Alg}(L, T, \text{all}, \text{all}) \). We assume that the operational semantics of the elementary commands is such that for an elementary test we have:

\[
\text{sp}(a) = \lambda x. \text{Alg}(\lambda x. (x \leq 100), \lambda x. (x > 100), \text{all}, \text{all})
\]

whereas for an elementary assignment \( e \) we have:

\[
\text{sp}(e) = \lambda x. \text{Alg}(\lambda x. (x \leq 100), \lambda x. (x > 100), \text{all}, \text{all})
\]

(Notice that for all \( \text{Alg}(L, T, \text{all}, \text{all}) \) the \( \text{sp}(S) \) is a complete join-morphism \((\text{i.e.}, \lambda x. \text{Alg}(\lambda x. (x \leq 100), \lambda x. (x > 100), \text{all}, \text{all})) \).

We assume that the operational semantics of the program \( \pi \) is such that at each program point \( i \in \{1, n\} \) the invariant assertion \( P_{i} \) which characterize the set of states which are the descendants of the input states satisfying a given entry assertion \( \Phi \) is the merge over all paths analysis of \( \pi \) using \( \text{sp} \) and \( \Phi \).

P is the least fixpoint of \( \Phi_{\pi}(\text{sp}, \Phi) \) of the system of equations \( P = F_{\pi}(\text{sp}, \Phi)(P) \) associated with the program \( \pi \) using \( \text{sp} \) and \( \Phi \).

270
Example 3.1.0.1

The system of forward semantic equations associated with the program 2.6.1 is:

\[
\begin{align*}
P_1 &= \phi \\
P_2 &= \mathcal{E}p((\lambda x . (x \leq 100))(P_1 \lor P_3)) \\
P_3 &= \mathcal{E}p((\lambda x . (x \geq 100))(P_1 \lor P_4)) \\
P_4 &= \mathcal{E}p((\lambda x . (x > 100))(P_2 \lor P_3))
\end{align*}
\]

taking \(\phi = \lambda x . (x = 1)\) its least fixpoint characterizes the descendants of the input states satisfying \(\phi\):

\[
\begin{align*}
P_1 &= \lambda x . (x = 1) \\
P_2 &= \lambda x . (1 \leq x \leq 100) \\
P_3 &= \lambda x . (25 \leq x \leq 100) \\
P_4 &= \lambda x . (x > 100)
\end{align*}
\]

End of Example.

3.2 Backward Semantics

The backward semantic analysis of a program consists in determining at each program point an invariant assertion which characterizes the set of states which are the ascendants of the output states satisfying a given exit specification \(\phi\).

Since we can consider the inverse of the state transition relation defined by the operational semantics of a new formalism is necessary in order to treat backward program analysis. Instead of Floyd's forward predicate transformer we just have to consider Hoare'[69]-Dijkstra's backward predicate transformer:

\[
\begin{align*}
Dp(q) &= \mathcal{E}p(A_{\lambda x . V . (P(X) \land x \geq \text{dom}(q) \land q(X))}) \\
Dp(a) &= \mathcal{E}p(A_{\lambda x . V . (x \geq \text{dom}(a) \land P(e(x)))})
\end{align*}
\]

(notice that \(\forall x \in L, Dp(S)\) is a complete join and meet morphism) and the inverted program graph \(G' = (n, E', n', \eta', \eta', C')\) where \(E' = \{(i, j) : (j, i) \in E\}, C' = \lambda x . \beta, j \beta E', \beta = \{(i, j)\}\).

Example 3.2.0.1

The inverted program graph corresponding to 2.6.1 is:

\[
\begin{align*}
&\mathcal{E}p((\lambda x . (x \leq 100))(P_1 \lor P_3)) \\
&\mathcal{E}p((\lambda x . (x > 100))(P_2 \lor P_3)) \\
&\mathcal{E}p((\lambda x . (x < 100))(P_2 \lor P_4)) \\
&\mathcal{E}p((\lambda x . (x = 1))(P_4))
\end{align*}
\]

The corresponding system of backward semantic equations is:

\[
\begin{align*}
P_1 &= Dp((\lambda x . (x \leq 100))(P_2 \lor P_3)) \\
P_2 &= Dp((\lambda x . (x > 100))(P_1 \lor P_4)) \\
P_3 &= Dp((\lambda x . (x < 100))(P_2 \lor P_4)) \\
P_4 &= \phi
\end{align*}
\]

The merge over all paths and largest fixpoint characterizations of the ascendants of the output states satisfying the exit specification \(\phi = \lambda x . (x = 101)\) are both equal to:

\[
\begin{align*}
P_1 &= \lambda x . (x \leq 101) \\
P_2 &= \lambda x . (x > 100) \\
P_3 &= \lambda x . (x \leq 101) \\
P_4 &= \phi = \lambda x . (x = 101)
\end{align*}
\]

End of Example.

In the following no distinction will be made between forward and backward program analyses because of the above mentioned symmetry.

4. APPROXIMATE ANALYSIS OF PROGRAMS

The semantic analysis of programs cannot be automated since neither the merge over all paths nor the least fixpoint characterization of the invariant assertions to be generated leads to a computable function. Therefore optimizing compilers and program verification systems are only concerned with the discovery of approximate invariants assertions. Here an approximate invariant assertion \(Q\) will be one which is implied by the exact invariant assertion \(P\) defined by the deductive semantics.

**DEFINITION 4.0.1**

If \(P, Q \subseteq A\) then "\(Q\) approximate \(P\)" iff \(P \Rightarrow Q\).

This definition of "approximate" is the one which is useful in logical analysis of programs, data type determination and data flow analysis (The dual one might be useful e.g. for proving termination).

The now classical lattice theoretic approach to approximate analysis of programs can be briefly sketched as follows: the representation of an approximate assertion is an element of a complete lattice \(\mathbb{A}(E, L, \top, \bot, \land, \lor, \Rightarrow)\). The meaning of the elements of \(\mathbb{A}\) is specified by a (too often implicit) order morphism \(\gamma\) mapping \(A\) to a subset of assertions \(\gamma(A) \subseteq A\). The intuition is that \(A\) is an implementable image of those aspects \(\gamma(A)\) of the program properties which are to be understood at each program point whereas the assertions belonging to \(A \setminus \gamma(A)\) are ignored (that is approximated from above in \(\gamma(A)\)). To each elementary command \(S\) is associated an isotope map \(t(S)\) from \(A\) to \(A\). The intent is that \(t(S)\) is an approximate predicate transformer such that \(t(S)(A)\) represents the propagation of the information \(t\) through the statement \(S\).

The ideal merge over all paths program-wide analysis [Graham & Wegman[76], Kam & Ulman[77], Rosen[78], Tarjan[79]] is often approximated by a fixpoint solution [Cousot & Cousot[77a], Jones & Muchnick[78], Kaplan & Ulman[78], Keilds[79], Tenenbaum[74]]. A fixpoint system of isotope equations \(\mathcal{F}(x)\) where \(F(x) (A' = A')\) is associated with the program graph. The approximate invariant assertions are generated by computing iteratively the least fixpoint of \(F\) starting from the infimum of \(A'\) and using any chaotic or asynchronous iteration strategy [Cousot[77a]] or the least fixpoint is approximated from above using an extrapolation technique in order to accelerate the convergence of the iterates whenever \(A\)
does not satisfy the ascending chain condition (Gouset & Gouret[77a]).

The design of $A, t, \gamma$, the implicit $\gamma$ and the de-
termination of the construction rules for $F$ are	often empirical. The correctness of the least fix-
point analysis is usually proved with respect to the
approximate merge over all paths analysis, the cor-
rectness of which is taken for granted. As opposed
to this empirical approach we now provide a formal
approach to the systematic design of an approximate
program analysis framework $(A, t, \gamma)$ given $(V, \delta, \tau)$
where $\tau$ is $sp$ for forward and $\omega$ for backward pro-
gram analyses.

5. DESIGN OF A SPACE OF APPROXIMATE ASSERTIONS

5.1 A Very Reasonable Assumption

Assume that for a specific-purpose analysis of
programs a subset $A_0A$ of assertions has been found
two semantic analyses (i.e., $sp(x:=x+y)(Q_1) =$
\[ \lambda(x, y), (x \leq y \wedge y \geq 0) \] and $sp(x:=x+y)(Q_2) = \lambda(x, y), (x \leq y \wedge y \geq 0)\) and next comparing them. Since these anal-
\[ \lambda(x, y), (x \leq y \wedge y \geq 0) \] whereas this is impossible with $Q_2$. On
the contrary the best choice is $Q_2$ for the program
$x := x_1, x := x_2$ since $sp(x := x_1, x := x_2)(Q_2) = \lambda(x, y),
(x \leq y \wedge y \geq 0)$ which implies $\lambda(x, y), (x \leq y \wedge y \geq 0)$ whereas
\[ sp(x := x_1, x := x_2)(Q_1) = \lambda(x, y), (x \leq y \wedge y \geq 0) \] does not
imply $\lambda(x, y), (x \leq y \wedge y \geq 0)$.

End of Example.

If any program must have an analysis which can
be approximated from above using $A$, and the process
for deriving the most useful approximate analysis of
any program is required to be deterministic then it is
reasonable to make the following:

ASSUMPTION 5.1.0.2

The set $A_0A$ of approximate assertions must be
chosen such that for all $P, A$ the set $(\lambda: F \Rightarrow 0)$ of
If the initial choice of $\bar{A}$ does not satisfy assumption 5.1.0.2 we can use the following:

**Theorem 5.2.0.4**

If $\bar{A} \in A$, the upper closure operator $\rho$ on $A$ such that $\rho(A)$ is the least Moore family containing $A$ is:

$$
\rho = \lambda A : \rho(A) \cup \{x \in A \mid \rho(x) = \text{true}\} : P \mapsto \emptyset
$$

$$
\rho(A) = \{S : S \subseteq (A \cup \{x \in A \mid \rho(x) = \text{true}\}) \land S \neq \emptyset\}
$$

**Example 5.2.0.5**

Returning to example 5.1.0.1 where $A = \{x \in B\}$ and $\bar{A} = \{\text{true}, \text{false}, 1, 0\}$ the least Moore family containing $\bar{A}$ is the one containing $1$, $\text{true}$, $\bar{A}$ and the meets of the non-empty subsets of $\bar{A}$ that is the complete lattice:

$$
\begin{array}{c}
\text{true} \\
1 = \text{false} \\
0 = \text{false}
\end{array}
$$

The corresponding approximation operator is:

$$
\rho = \lambda P : \rho(A) \cup \{x \in A \mid \rho(x) = \text{true}\} : P \mapsto \emptyset
$$

$$
\rho(A) = \{S : S \subseteq (A \cup \{x \in A \mid \rho(x) = \text{true}\}) \land S \neq \emptyset\}
$$

End of Example.

**5.3 Representation of the Lattice of Approximate Assertions**

In order to represent the approximate assertions in a computer memory we must use a complete lattice $A(\xi,\mu,\eta,\eta)$ such that the similar algebra $\bar{A}$ is the least Moore family containing $A$ and $A(\xi,\mu,\eta,\eta)$ is isomorphic. Let $\gamma \in (A \rightarrow \bar{A})$ be the corresponding lattice isomorphism. Let $\alpha \cdot (A \rightarrow \bar{A})$ be $\gamma^{-1} \circ \rho$. $\alpha(P)$ is the representation of the least upper approximation of the assertion $P$ whereas $\gamma(G)$ provides the meaning of $G$.

**Definition 5.3.0.1**

Let $L_1(E_1)$ and $L_2(E_2)$ be posets. $\langle \alpha, \gamma \rangle$ is a pair of adjacent functions if and only if:

- $\alpha : \{L_1 \rightarrow L_2\}$ is isotone
- $\gamma : \{L_2 \rightarrow L_1\}$ is isotone
- $\forall x \in L_1, \forall y \in L_1 : (x \leq_1 y) \Rightarrow (\alpha(x) \leq_2 \gamma(y))$

(Contrary to Scott[72]'s definition, $L_1$ and $L_2$ are not required to be continuous lattices and $\alpha, \gamma$ need not be continuous).

**Theorem 5.3.0.2**

If $\rho$ is an upper closure operator on $\bar{A}$, the image $\gamma(A)$ of $A(\xi,\mu,\eta,\eta)$ through the lattice isomorphism $\gamma$ is equal to $\rho(A) = \rho(\{\text{false}, 1, \text{false}\})$.

$$\lambda x, (\rho(\{x\}) \land \alpha \gamma)$$

$$- \langle \alpha, \gamma \rangle$$

- $\alpha$ is onto, $\gamma$ is one-to-one

Reciprocally the approximation process can be defined by the lattice $A(\xi,\mu,\eta,\eta)$ and a pair of adjacent functions. Such a pair $\langle \alpha, \gamma \rangle$ defines a Galois connection between $A$ and the dual of $A$.

**Definition 5.3.0.3**

Let $L_1(E_1)$ and $L_2(E_2)$ be posets. $\alpha : \{L_1 \rightarrow L_2\}$ and $\gamma : \{L_2 \rightarrow L_1\}$ are the pair of adjacent functions if and only if:

1. $\alpha$ is isotone $\forall x \in L_1, \forall y \in L_2, (x \leq_1 y) \Rightarrow (\alpha(x) \leq_2 \gamma(y))$
2. $\gamma$ is isotone $\forall x \in L_2, \forall y \in L_1, (x \leq_2 y) \Rightarrow (\gamma(x) \leq_1 \alpha(y))$
3. $\forall x \in L_1, \forall y \in L_2, (x \leq_1 y) \Rightarrow (\alpha(x) \leq_2 \gamma(y))$
4. $\forall x \in L_2, \forall y \in L_1, (x \leq_2 y) \Rightarrow (\gamma(x) \leq_1 \alpha(y))$

The above conditions (3) and (4) are equivalent to:

$$\forall x \in L_1, \forall y \in L_2, (x \leq_1 y) \Rightarrow (\alpha(x) \leq_2 \gamma(y))$$

(Birkhoff[57]), hence we have:

**Theorem 5.3.0.4**

Let $L_1(E_1)$ and $L_2(E_2)$ be posets. $\alpha : \{L_1 \rightarrow L_2\}$ and $\gamma : \{L_2 \rightarrow L_1\}$ are the pair of adjacent functions if and only if:

$$\forall x \in L_1, \forall y \in L_2, (x \leq_1 y) \Rightarrow (\alpha(x) \leq_2 \gamma(y))$$

Theorem 5.3.0.4, Ore[44, Th.2] and Pickert[52] imply:

**Corollary 5.3.0.5**

Let $L_1(E_1)$ and $L_2(E_2)$ be posets. $\alpha : \{L_1 \rightarrow L_2\}$ and $\gamma : \{L_2 \rightarrow L_1\}$ are the pair of adjacent functions if and only if:

1. $\alpha$ is an upper closure operator on $L_1$, $\gamma$ is a lower closure operator on $L_2$.
2. $\alpha$ and $\gamma$ are isotone.
3. $\forall x \in L_1, \forall y \in L_2, (x \leq_1 y) \Rightarrow (\alpha(x) \leq_2 \gamma(y))$
4. $\forall x \in L_2, \forall y \in L_1, (x \leq_2 y) \Rightarrow (\gamma(x) \leq_1 \alpha(y))$

Moreover if $L_1(E_1)$ and $L_2(E_2)$ are complete lattices then:

1. $\gamma(\alpha(L_1))$ and $\alpha(\gamma(L_2))$ are complete lattices.
2. $\alpha$ and $\gamma$ are isotone.
3. $\forall x \in L_1, \forall y \in L_2, (x \leq_1 y) \Rightarrow (\alpha(x) \leq_2 \gamma(y))$
4. $\forall x \in L_2, \forall y \in L_1, (x \leq_2 y) \Rightarrow (\gamma(x) \leq_1 \alpha(y))$

In complement we will need the following:

**Theorem 5.3.0.6**

Let $L_1(E_1, L_2, L_3, L_4, L_5)$ and $L_2(E_2, L_3, L_4, L_5, L_6)$ be complete lattices and $\alpha : \{L_1 \rightarrow L_2\}$ and $\gamma : \{L_2 \rightarrow L_1\}$ are the pair of adjacent functions if and only if:

1. $\alpha$ is onto (surjective) if and only if $\gamma$ is one-to-one (injective).
2. $\alpha$ and $\gamma$ are isotone.
\[ \gamma = \lambda x \in L_1. \lambda y \in L_2. \beta(\alpha(x), y) \]
\[ \alpha \] is an isomorphism from the complete lattice \( \gamma(A_1) \) onto the complete lattice \( L_2 \) the inverse of which is \( \gamma \).
(3) \( \alpha \) is one-to-one if and only if \( \gamma \) is onto and if and only if \( \gamma \circ \alpha = \lambda x \in L_1. (x) \)

We use the notation \( L_1 \triangleleft \alpha, \gamma \triangleright L_2 \) to state that \( L_1 \) and \( L_2 \) are connected by the pair \( \alpha, \gamma \) of adjoint functions which are respectively surjective and injective. If \( \alpha \) is a complete join-morphism from \( L_1 \) onto \( L_2 \) (respectively \( \gamma \) is a one-to-one complete meet-morphism from \( L_2 \) into \( L_1 \) we write \( L_1 \triangleleft \alpha, \gamma \triangleright L_2 \) \( L_1 \triangleleft \alpha, \gamma \triangleright L_2 \) and assume that the adjoint \( \gamma(A) \) is determined by \( 5.3.0.5.3(2), 5.3.0.5.3(1) \).

In the literature the most usual method for defining a program analysis framework is to specify the complete lattice \( \Lambda(E_1, L_1, L_2) \) representing approximate assertions and to informally describe the meaning of its elements (e.g., constant propagation, Kildall[73], Kam & Sulman[71]). Hence the function \( \gamma \in (A \rightarrow A) \) remains implicit.

It is often the case that \( \gamma \) is only assumed to be a (complete) join-semi-lattice \( \Lambda(E_1, L_1, L_2) \) or a dual meet-semi-lattice for some authors) but since an infimum is adjoined to \( A \) it is in fact a complete lattice (even when the meet-operation is not used or what is called meet is not \( \sqcap \) e.g., Wegbreit[75]).

When \( \gamma \in (A \rightarrow A) \) is isotope but not a complete meet-morphism the set \( \gamma(A) \) does not fulfill assumption 5.1.0.2 with the consequences examined at paragraph 5.1. The design of \( \gamma(A) \) and \( A \) can be revised as stated by theorem 5.2.0.4.

When \( \gamma \in (A \rightarrow A) \) is a complete meet-morphism but not one-to-one, several distinct elements of \( A \) have the same meaning. Since this is useless, the design of \( A \) and \( \gamma \) can be revised as follows:

**THEOREM 5.3.0.7**

Let \( \Lambda(E_1, L_1, L_2) \) be a complete lattice and \( \gamma \in (A \rightarrow A) \) be a complete meet-morphism. Let \( \sigma \in (A \rightarrow A) \)
\( \lambda x \in \Lambda(\gamma(x), \gamma(y)) \), \( \Lambda = \sigma(A) \), \( \gamma = \gamma(A) : \)

- \( \forall x \in A \), \( \gamma(x) = \gamma(\sigma(x)) \)
- \( \sigma \) is a lower closure operator on \( A \)
- \( \gamma \) is a one-to-one complete meet-morphism from the complete lattice \( \Lambda(E_1, L_1, L_2) \) into \( A \)

Since \( \gamma(A) \) is \( \gamma(A) \), \( A \) and \( A \) have the same expressive power. Among all subsets of \( A \) which have the expressive power of \( A \), \( A \) is one with minimal cardinality.

**THEOREM 5.3.0.8**

(1) \( \forall x \in A \), \( \gamma(x) = \gamma(\sigma(x)) \)
(2) \( \forall x \in A \)
(3) \( \forall x \in A \), \( \gamma(x) = \gamma(\sigma(x)) \)
(4) \( \forall x \in A \), \( \gamma(x) = \gamma(\sigma(x)) \)
(5) \( \forall x \in A \), \( \gamma(x) = \gamma(\sigma(x)) \)

6. EQUIVALENT METHODS FOR SPECIFYING A SPACE OF APPROXIMATE ASSERTIONS

A space of approximate assertions can be specified either by a Moore family or by an upper closure operator. Moore families can be characterized using definition 5.1.0.2 or theorem 5.1.0.3 and 5.2.0.4. In addition to theorems 5.2.0.2(1) and 5.3.0.8 we now study and exemplify various equivalent methods which can be used to define an upper closure operator.

### 6.1 Least Closure Operator Greater than or Equal to an Arbitrary Function

**THEOREM 6.1.0.1**

Let \( \Lambda(E_1, L_1, L_2) \) be a complete lattice and \( f \in (L \rightarrow L) \).
- \( \lambda x \in L_1 \rightarrow \Lambda(\lambda y \in L_2 \rightarrow \Lambda(f(y), y \in x)) \)
- \( \lambda y \in L_2 \rightarrow \Lambda(f(y), y \in x) \)
- \( \lambda y \in L_2 \rightarrow \Lambda(f(y), y \in x) \)
- \( \lambda y \in L_2 \rightarrow \Lambda(f(y), y \in x) \)

6.2. Definition of a Space of Approximate Assertions by Composition of Upper Closure Operators

The composition of two upper closure operators on \( A \) is usually not a closure operator (Cousot & Reif[74]). However the space of approximate assertions can be designed by successive approximations using the following composition of upper closure operators:

**THEOREM 6.2.0.1**

Let \( \Lambda(E_1, L_1, L_2) \) be a complete lattice, \( p \) an upper closure operator on \( A \) and \( \eta \) an upper closure operator on \( \eta(p) \). Then \( \eta(p) \) is an upper closure operator on \( A \) and \( \eta(p) \).
7. DESIGN OF THE APPROXIMATE PREDICATE TRANSFORMER
According to theorem 7.2.0.4 the best upper approximation of $ap(\lambda x y, y(\lambda z y))$ in $A$ is $\alpha ap(\lambda x y, y(\lambda z y)) = \alpha ap(\lambda x y, y(z)) = \cup y z = \lambda x y z$. If $P(a) = \lambda a' \in P P(a)$ then $t(\lambda) = \lambda a' \in P P(a)$. Also $\alpha ap(\lambda x y, y(z)) = \cup y z = \lambda x y z$. Therefore, the best upper approximation of $ap(\lambda x y, y(\lambda z y))$ in $A$ is $\alpha ap(\lambda x y, y(\lambda z y)) = \alpha ap(\lambda x y, y(z)) = \cup y z = \lambda x y z$. The same way $\alpha ap(\lambda x y, y(z)) = \cup y z = \lambda x y z$. Therefore, the best upper approximation of $ap(\lambda x y, y(\lambda z y))$ in $A$ is $\alpha ap(\lambda x y, y(z)) = \cup y z = \lambda x y z$.

End of Example.

Example 7.2.0.6

Some program analyses (such as "reaching definitions", "available expressions", "live variables", ... [Aho & Ullman, 77]) are "history sensitive" because the approximate assertions which are useful at each program point $p$ characterize sets of sequences of states (for execution paths from the entry point to $p$) and not sets of states. In such a case, Hoare [78] formal definition of languages using sets of sequential traces is more convenient that the deductive semantics of paragraph 3.

7.2.0.6.1. Associating a Set of Traces with a Program

Given a Universe $V$ of values, a set $I_0$ of elementary assignments, a set $L_0$ of elementary tests, a set of sequential traces $T_1$, the free monoid $I_0 \cup L_0$.

The concatenation operation "$\triangleright$" is extended to elements of the complete lattice $T_0 = (\alpha \cup T_0, V, N)$ by $\nabla_t = (\rightarrow \cup I_0, V, N)$.

Let us define a forward "set of tracks transformer" $\nabla_t \in L_0 \cup \nabla_t$ as $\lambda \nabla_t \cup (I_0 \cup \nabla_t)$. The set of traces associated with a program $T$ and an entry specification $\Phi E \in L_0 \cup \Phi E$ is $\Phi E \cap L_0 \cup \Phi E$.

7.2.0.6.2. Approximating a Set of Traces by an Algebra Characterizing the Descendants of the Entry State

The connection with the deductive semantics of paragraph 3 is made using $a(\lambda) = \lambda x \in V, x \in N$ and $a(\lambda) = \lambda x \in V, x \in N$ such that for any set $T$ of traces, $a(\lambda)$ characterizes the possible descendants of the entry states (belonging to $V$). The set of traces executed are $\alpha ap(\lambda)$. From an obvious observation (as noted in Section 3.2) we shall have $\alpha ap(\lambda) = \alpha ap(\lambda)$, $\alpha ap(\lambda) = \alpha ap(\lambda)$, and $\alpha ap(\lambda) = \alpha ap(\lambda)$. Therefore, the best upper approximation of $ap(\lambda)$ in $A$ is $\alpha ap(\lambda)$.

Since $a(a) = \alpha ap(\lambda)$, $\alpha ap(\lambda) = \alpha ap(\lambda)$, and $\alpha ap(\lambda) = \alpha ap(\lambda)$, we have $\alpha ap(\lambda) = \alpha ap(\lambda)$, $\alpha ap(\lambda) = \alpha ap(\lambda)$, and $\alpha ap(\lambda) = \alpha ap(\lambda)$. Theorem 7.3.0.6.2.1. Hence, the best upper approximation of $ap(\lambda)$ in $A$ is $\alpha ap(\lambda)$.

Since $\alpha ap(\lambda) = \alpha ap(\lambda)$, $\alpha ap(\lambda) = \alpha ap(\lambda)$, and $\alpha ap(\lambda) = \alpha ap(\lambda)$, we have $\alpha ap(\lambda) = \alpha ap(\lambda)$, $\alpha ap(\lambda) = \alpha ap(\lambda)$, and $\alpha ap(\lambda) = \alpha ap(\lambda)$. Hence, the best upper approximation of $ap(\lambda)$ in $A$ is $\alpha ap(\lambda)$.

In order to briefly illustrate the hierarchy of program analysis frameworks, let us consider three comparable examples: the approximation function of which can be sketched using a geometrical analogy. Let $P$ be a predicate over two numerical variables $x$ and $y$ the characteristic set of which is the following:

$$ y = p(x, y) $$

$$ y = f(x, y) $$

$$ y = g(x, y) $$

8. Hierarchy of Program Analysis Frameworks

Once the semantics of programs has been defined by $(A, L, \gamma)$ all program analysis frameworks $(A, L, \gamma)$ are specified up to the isomorphism $\gamma$ by $(p(A), \alpha, \gamma)$ where $p(A) = \alpha$ is an upper closure operator on $A$ and $A = \alpha A$. Program analysis frameworks can be partially ordered using the ordering of the corresponding closure operators on $A$ since whenever $p_1 \subseteq p_2$, $p_2 \subseteq p_1$ are so that program analysis frameworks corresponding to $p_1$ yield sharper information than the one corresponding to $p_2$ (whichever global program analysis method is used). The following theorem is a constructive version of Ward [42, §7.3.5]:

**Theorem 8.0.1**

The set of upper closure operators on a complete lattice $(\ell, \ell, \ell, \ell, \ell, \ell)$ is a complete lattice $(\ell, \ell, \ell, \ell, \ell, \ell) = \ell$ of $(\ell, \ell, \ell, \ell, \ell, \ell) = \ell$.
The upper closure operator of example 5.2.0.5 defines a very rough approximation consisting in approximating this set by the quarter of plane containing all its points:

\[ y = \rho_p(x, y) \]

A more precise approximation (example 6.3.0.5) consists in approximating the characteristic set of \( P \) by the smallest rectangle including it and whose sides run parallel with the axes:

\[ y = \rho_p(x, y) \]

A refinement consists in approximating the characteristic set of \( P \) by its convex-hull:

\[ y = \rho_p(x, y) \]

The corresponding framework was used for the automatic discovery of linear constraints among variables of programs ([Corson & Hallwachs '78]).

End of Example.

9. MERGE OVER ALL PATHS VERSUS LEAST FIXPOINT GLOBAL ANALYSIS OF PROGRAMS

9.1 "Distributive" Program Analysis Frameworks

We recalled at paragraph 4 that once a program analysis framework \((A, t, y)\) has been designed, the program-wide analysis problem has various solutions including the merge over all paths and least fixpoint solutions. It is known (Kam & Ullman '77) that when \( A \) satisfies the ascending chain condition and \( \forall \sigma \in L, t(S) \) is isometric we have \( \text{MOP}_A(t, \sigma) = \text{lfp}(F^*_\tau(t, \sigma)) \).

The additional hypothesis that \( \forall \sigma \in L, t(S) \) is a join-morphism (sometimes called join-distributive map) implies \( \text{MOP}_A(t, \sigma) = \text{lfp}(F^*_\tau(t, \sigma)) \).

Slightly more general is the following:

**Theorem 9.1.0.1**

If \((A, t, \sigma, \mu, \nu)\) is a complete lattice and \( t: (L \rightarrow (A^+ A)) \) is such that \( \forall \sigma \in L, t(S) \) is isometric then for all programs \( \nu \) and \( \phi \in A, \text{MOP}_A(t, \sigma) = \text{lfp}(F^*_\tau(t, \sigma)) \).

This theorem is implicitly used at paragraph 3 taking \( A = (V = B) \rightleftharpoons (\lambda x: f(x,a), \lambda x: t(x, a), \lambda x: v, \lambda v) \) for \((A, t, \tau, \mu, \nu)\) and either \( \phi = \phi \) or \( \phi \) for \( t \).

If \( A = (A, \leq, \tau) \) and \( t: (L \rightarrow (A^+ A)) \) then the above theorem establishes the correctness of \( \text{lfp}(F^*_\tau(t, \sigma)) \) with respect to \( \text{MOP}_A(t, \sigma) \). In the literature the correctness of \( \text{MOP}_A(t, \sigma) \) is generally taken for granted. Also \( \text{MOP}_A(t, \sigma) \) is considered as the desired solution to program-wide analysis problems since whenever some \( t(S) \) is not a complete join-morphism \( \text{MOP}_A(t, \sigma) \) can be strictly better than \( \text{lfp}(F^*_\tau(t, \sigma)) \).

When \( A \) satisfies the ascending chain condition \( \forall \sigma \in L, t(S) \) is isometric, which is not necessarily the case of \( \text{MOP}_A(t, \sigma) \). In that case a variety of methods can be used (e.g., Rosen '78) which can find sharper information that fixpoint methods and therefore approach the ideal merge over all paths solution which provides the maximum information relevant to \( A, t \) and \( Y \).

In our opinion the above argument is not entirely convincing since for different correct approximate predicate transformers \( t_1, t_2 \in (L \rightarrow (A^+ A)) \) it may be the case that \( \text{lfp}(F^*_\tau(t_1, \sigma)) \neq \text{MOP}_A(t_2, \sigma) \). In order to relieve from the burden of badly chosen approximate predicate transformers the argument must consider the best approximate predicate transformer relevant to \( A \) (Theorem 7.2.0.4). Then the following result is a useful complement to Theorem 9.1.0.1:

**Theorem 9.1.0.2**

Let \( \tau: (L \rightarrow (A^+ A)) \) be the best correct upper approximation of \( \tau: (L \rightarrow (A^+ A)) \) in \( \text{MOP}_A(t, \sigma) \). If \( \text{MOP}_A(t, \sigma) \) is a complete sublattice of \( A \) then \( \text{MOP}_A(t, \sigma) = \text{lfp}(F^*_\tau(t, \sigma)) \).

**Example 9.1.0.3**

If \( A = (Z = B) \) and \( \tau = \gamma(A) \) where:

\[ A = \tau \]

and \( \gamma = \lambda u: \mu \gamma \mu \gamma \), \( \gamma^- = \lambda u: \mu \gamma \mu \gamma \), \( \gamma^+ = \lambda u: \mu \gamma \mu \gamma \), \( \gamma^0 = \lambda u: \mu \gamma \mu \gamma \), etc. then \( A \) is not a sublattice of \( A \) since \( \gamma^+ \gamma \gamma^+ \gamma \gamma \) is not an element of the loop. The merge over all paths analysis of the program:

\[ \text{if } x > 0 \text{ then while } x > 0 \text{ do } x :=-x \text{ od; j} \]

which is powerful enough in order to determine that the while-loop does not terminate is strictly better than the least fixpoint analysis (which fails to discover that \( j \) is invariant on the exit path of the loop).

End of Example.

9.2 "Non-Distributive" Program Analysis Frameworks

The merge over all paths analysis of a program using some "non-distributive" program analysis framework can always be defined by means of the least fixpoint of a system of isometric equations associated with that program:

**Theorem 9.2.0.1**

Let \((A, t, \sigma, \mu, \nu)\) be a complete lattice, \( \text{MOP}(L \rightarrow (A^+ A)) \) be an approximate predicate transformer, \( 2^A \) be the complete lattice of all subsets of \( A \), \( \tau: (L \rightarrow (2^A \rightarrow 2^A)) \) be \( \lambda \sigma: (A, \mu: (\text{lfp}(F^*_\tau(t, \sigma)) \rightarrow \text{MOP}_A(t, \sigma))) \) and \( \text{MOP}(L \rightarrow (2^A \rightarrow 2^A)) \) be \( \lambda \nu: (2^A \rightarrow 2^A) \).

- \( \forall \sigma \in L, t(S) \) is a complete \( \nu \)-morphism
- \( \forall \nu, \forall \sigma: A \mapsto (\text{lfp}(F^*_\tau(t, \sigma))) = \text{MOP}_A(t, \sigma) \)

The above construction is not fully satisfactory since \( (2^A, \tau) \) is not isomorphic to \((A, t)\) when \( t \) is a complete join-morphism, so that the choice of \((2^A, \tau)\) in order to define \( \text{MOP}_A(t, \sigma) \) as a least
10. COMBINATION OF PROGRAM ANALYSIS FRAMEWORKS

The ideal method in order to construct a program analyzer (to be integrated in optimizing compilers or program verification systems) would consist in a separate design and implementation of various complementary program analysis frameworks which could then be systematically combined using a once for all implemented assembler. In this section, we show that such an automatic combination of independently designed parts would not lead to an optimal analyser and that unfortunately the efficient combination of program analysis frameworks often necessitates the revision of the original design phase.

10.1 Reduced Cardinal Product of Program Analysis Frameworks

**Theorem 10.1.0.1**

Let \((A_1, \tau_1, \gamma_1), (A_2, \tau_2, \gamma_2)\) be two program analysis frameworks such that \(A_1 \leq \gamma_1 \leq A, A_2 \leq \gamma_2 \leq A\) and \(\tau_1, \tau_2\) are correct upper approximations of \(\tau\) in \(A_1, A_2\). The direct product \((A \times \gamma)\) of \((A_1, \gamma_1)\) and \((A_2, \gamma_2)\)
Remark 10.1.0.4

Let $L_1(E_1)$, $L_2(E_2)$ be posets. The cardinal sum of $L_1$ and $L_2$ is the set of all elements in $L_1$ or $L_2$, considered as disjoint. When $L_1(E_1,T_1,\lambda_1,\mu_1)$ and $L_2(E_2,T_2,\lambda_2,\mu_2)$ are complete lattices we can define the disjoint sum $L_1 \cup L_2 \cup \{t,T\}$ with ordering $x \leq y$ iff $(x=t)$ or $(y=t)$ or $(x \in L_1, y \in E_2)$ or $(x \in L_2, y \in E_1)$. The meaning of elements of $L_1 \cup L_2$ can be defined as $Y(j) = \lambda x, \chi(x) \rightarrow \lambda y \in y \mapsto y \in y$. Even when $\gamma_1$ and $\gamma_2$ are one-to-one complete meet-morphisms, $\gamma$ may be neither one-to-one nor a complete meet-morphism. In order to satisfy assumption 5.1.0.2 the set $L_1 \cup L_2$ must be completed using theorem 2.2.0.4. Then it turns out that the least Moore family containing $\gamma_1,\gamma_2 \in L_1 \cup L_2$ is equal to $\gamma'(L_1 \cup L_2)$ (as defined in theorem 10.1.0.2). Therefore the use of disjoint sums amounts to the use of reduced products.

End of Remark.

10.2 Reduced Cardinal Power of Program Analysis Frameworks

The cardinal power $L_1^{|X|}$ with base $L_2(E_2,T_2,\lambda_2,\mu_2)$ and exponent $L_1(E_1,T_1,\lambda_1,\mu_1)$ (hereafter noted $f(a) = L_1 \times L_2$) is the set of all iso-tonic maps from $L_1$ to $L_2$ with the following property: for any $x \in L_1$, there is a unique function $f$ such that $f(x) = y \in L_2$ for all $y \in L_2$. Two program analysis frameworks $(A_1,E_1,Y_1)$ and $(A_2,E_2,Y_2)$ can be combined by letting $g = f(\alpha_0) + g_1$ mean that for all $x \in A_1$, $g(x) \in y \in y_1 \in y_2$ holds whenever $y_1(x) \in y_2(x)$ holds.

Theorem 10.2.0.1

The reduced cardinal power with base $(A_2,T_2,Y_2)$ and exponent $(A_1,T_1,\gamma_1)$ is $(A_2,T_2,\gamma_2)$ where $\gamma_2 = \lambda \alpha(T_1) \gamma_1(x) \gamma_2(y) \gamma_2(z)$.

Example 10.2.0.2

According to theorem 10.1.0.1 the direct product of the above analyses cannot yield sharper information.
using the reduced cardinal product of $A_1$ and $A_2$ yields no information since no relationship can be discovered between $b$ and $x$.

Following theorem 10.2.0.1 we determine that if
\[ g(x_1) \rightarrow A_2 \text{ then } Y(g) = Y(y_1) \rightarrow A_2 \text{ or } Y(g) = Y(y_1) \rightarrow A_2 \text{ or } Y(g) = Y(y_1) \rightarrow A_2 \text{ or } Y(g) = Y(y_1) \rightarrow A_2 \]
Therefore $g(x) = g(y_1)$ if and only if $h(x_1) = h(y_2)$ or $h(y_2) = g(x)$. It follows that $g(\{a_2|a_1\})$ is isomorphic to $\{t_1|t_2\}$ for $A_1 \times A_2$.

The system of equations associated with the upper program and the entry specification $\lambda b. f_1 f_2$ is then:

\[
\begin{align*}
g_1 &= \lambda b. f_1 f_2 \text{ b then } g_1 \text{ else } g_2 \\
g_2 &= \lambda b. f_1 f_2 \text{ b then } g_1 \text{ else } g_2 \\
g_3 &= \lambda b. \text{deaf}(g_1(b)) \\
g_4 &= \lambda b. f_1 f_2 \text{ b then } (g_1(b) \rightarrow f_1) \text{ else } g_2 \\
g_5 &= \lambda b. f_1 f_2 \text{ b then } g_1(b) \rightarrow g_2 \\
\end{align*}
\]

where $\text{deaf}(g_1(b)) = \lambda b. (g_1(b) \rightarrow f_1) \text{ else } g_2$

End of Example.

11. REFERENCES


Ore O. [1943], Combination of closure relations, Ann. of Math., 44 (1943), 514-533.

Ore O. [1944], Galois connections, Trans. AMS, 55 (1944), 499-515.


Scott D. [1976], Data types as lattices, SIAM Comp. 5, (3Sept. 1976), 522-567.


