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
SPECIAL INTEREST GROUP ON AUTOMATA AND COMPUTABILITY THEORY
SPECIAL INTEREST GROUP ON PROGRAMMING LANGUAGES
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, \eta, \gamma) where A is a lattice of (approximate) assertions, \eta is an (approximate) predicate transformer and \gamma 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.

Pre liminary definitions are given in Section 2 concerning the merge over all paths and (least) fixpoint program-wide analysis methods. In Section 3 we briefly define the (forward and backward) deductive semantics of programs which is later used as a formal basis in order to prove the correctness of the approximate program analysis frameworks. Section 4 very shortly recall the main elements of the lattice theoretic approach to approximate semantic analysis of programs.

The design of a space of approximate assertions A is studied in Section 5. We first justify the very reasonable assumption that A must be chosen such that the exact invariant assertions of any program must have an upper approximation in A and that the approximate analysis of any program must be performed using a deterministic process. These assumptions are shown to imply that A is a Moore family, that the approximation operator (which defines the least upper approximation of any assertion) is an upper closure operator and that A is necessarily a complete lattice.

Next we show that the connection between a space of approximate assertions and a computer representation is naturally made using a pair of lattice adjoint functions. This type of connection between two complete lattices is related to Galois connections thus making available classical mathematical results. Additional results are proved, they hold when no two approximate assertions have the same meaning.

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 isotomy is not required for approximate predicate transformers to be correct it is shown that non-isotone program analysis frameworks are manageable although it is later argued that the isotony hypothesis is natural. We next show that among all possible approximate predicate transformers which can be used with a given space of approximate assertions there exists a best one which provides the maximum information relative to a program-wide analysis method. The best approximate predicate transformer induced by a space of approximate assertions turns out to be isotone. Some interesting consequences of the existence of a best predicate transformer are examined. One is that we have in hand a formal specification of the programs which have to be written in order to implement a program analysis framework once a representation of the space of approximate assertions has been chosen. Examples are given, including ones where the semantics of programs is formalized using Hoare[71]'s sets of traces.

In Section 8 we show that a hierarchy of approximate analyses can be defined according to the fineness of the approximations specified by a program analysis framework. Some elements of the hierarchy are shortly exhibited and related to the relevant literature.

In Section 9 we consider global program analysis methods. The distinction between "distributive" and "non-distributive" program analysis frameworks is studied. It is shown that when the best approximate predicate transformer is considered the coincidence or not of the merge over all paths and least fixpoint global analyses of programs is a consequence of the choice of the space of approximate assertions. It is
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 isotope 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 \( V \) is a program graph and \( V \) is the universe in which the program variables take their values.

The set of elementary commands consists in

where \( \text{path}(i) \) is the set of paths from the entry point \( n_1 \) to the vertex \( i \) and \( \tilde{\tau} \in (E^* \rightarrow \{A\rightarrow A\}) \) is recursively defined as follows: if \( p \) is an empty path then \( \tilde{\tau}(p) \) is the identity map on \( A \) else \( p = (q,s) \) where \( q \in E^* \), \( \alpha \in E \) and \( \tilde{\tau}(p) = \lambda \phi.t[C(\alpha)](\tilde{\tau}(q))(\phi) \).

The system of equations \( P = F(\tau,\phi)(P) \) associated with the program \( \pi \) using \((A,\tau)\) and \( \phi \) is defined as follows:

\[
\begin{align*}
P_1 &= \phi \\
P_j &= \bigcup_{i \in \text{pred}(j)} t[C(i,j)](P_i) \quad \text{if } j \in [1,n]-\{n_1\}
\end{align*}
\]

Notation: If \( M(m,1,\tau,\lambda,x) \) is a complete lattice then the set \((L \rightarrow M)\) of total maps from the set \( L \) into \( M \) is a complete lattice \((L \rightarrow M)(\delta,\lambda,\tau,\lambda',\lambda''\equiv')\) for the pointwise ordering \( f \leq g \iff \forall x \in L, f(x) \leq g(x) \).

In the following the distinction between \( S,\downarrow,\lor,\downarrow \),
Example 3.1.0.1

The system of forward semantic equations associated with the program 2.0.1 is:

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

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

\[
\begin{align*}
P_1 &= [\lambda x.(x = 1)] \\
P_2 &= [\lambda x.(1 \leq x \leq 100)] \\
P_3 &= [\lambda x.(2 \leq x \leq 101)] \\
P_4 &= [\lambda x.(x > 101)]
\end{align*}
\]

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 descendants 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 no new formalism is necessary in order to treat backward program analysis. Instead of Floyd's forward predicate transformer we just have to consider Hoare[66]-Dijkstra[73]'s backward predicate transformer:

\[
\begin{align*}
wp[q] &= \text{wp}(\bar{A},[\lambda x.V, (P(x) \land x < \text{dom}(q) \land q(x))]) \\
wp[a] &= \text{wp}(\bar{A},[\lambda x.V, (x \text{dom}([a] \land P(x))])
\end{align*}
\]

(notice that $\forall \in L, \wp(\emptyset)$ is the complete join and meet morphism) and the inverted program graph $G' = (L', E', n_0, n_f, C')$ where $E' = \{(i, j) : [i, j] E \cup \{(i, j) \in E, C' = \lambda (i, j) E, \lambda C \leq E \}
\]

Example 3.2.0.1

The inverted program graph corresponding to 2.0.1 is:

\[
\begin{align*}
\lambda x.(x \leq 100) & \quad \lambda x.(x > 101)
\end{align*}
\]

both equal to:

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

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 $P, Q \in A$ 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 $[A, \leq, \bot, \top]$. The meaning of the elements of $A$ is specified by a (too often implicit) order morphism $Y$ mapping $A$ to a subset of assertions $\hat{A} = Y(A) \subseteq \hat{A}$. The intuition is that $A$ is an implementable image of those aspects $Y(A)$ of the program properties which are to be understood at each program point whereas the assertions belonging to $\hat{A} \setminus Y(A)$ are ignored (that is approximated from above in $Y(A)$). To each elementary command $S$ in $L$ is associated an isotope map $t(S)$ such that $t(S)(A) = \hat{A}$.
5. DESIGN OF A SPACE OF APPROXIMATE ASSERTIONS

5.1 A Very Reasonable Assumption

Assume that for a specific-purpose analysis of a program a subset $A$ of assertions has been found to provide meaningful information.

Since any invariant assertion $P \in A$ for any program must have an upper approximation $Q$ in $A$, the set $(Q, \rightarrow, P \rightarrow Q)$ must be non-empty.

Let $P \in A$ be an assertion and assume that we want to analyze a program $\pi$ using the merge over all paths semantic analysis and an extension condition $Q$ which is an upper approximation of $P$ in $A$. What is the best choice for $Q$? It is clear that if $P \rightarrow Q' \rightarrow Q$ then $P \rightarrow Q \rightarrow Q$ and by isotony the analysis $I(P) = I(P, (Q', Q))$ is more precise than $I(P, (Q, Q))$.

Hence $Q$ must be a minimal upper approximation of $P$ in $A$ (that is such that $(P \rightarrow Q' \rightarrow Q : P \rightarrow Q' \rightarrow Q)$) $\subseteq (P \rightarrow Q' \rightarrow Q : P \rightarrow Q \rightarrow Q)$). Assume that the set $U$ of minimal upper approximations of $P$ in $A$ has a cardinality greater than 1. What is the "best" possible choice for $Q$ in $U$? If $Q_1, Q_2 \in U$ and $Q_1 \neq Q_2$ then $Q_1$ and $Q_2$ are not necessarily comparable so that $I(P, (Q_1, Q_2))$ and $I(P, (Q_1, Q_2))$ cannot be defined using the precision criterion provided by the ordering $\rightarrow$. The only way to determine which of the two alternatives will be the most useful in order to answer a given set of application dependent questions about the program is to try both of them. Also the best choice may vary from one program to another. This try and see choice method leads to a non-deterministic analysis method which is unacceptable because of obvious efficiency considerations. Therefore it is reasonable to choose $\bar{A}$ such that Card($\bar{A}$) = 1.

Example 5.1.0.1

Assume that $A = (\{z \times z\}) + B$ where $Z$ is the set of integers and $\bar{A} = \{A_{xy} : P(x) \land P(y)\}$ (for $P, P_1 \subset \{A_{xyz}, A_{xyz}, A_{xyz}, A_{xyz}\}$). The assertion $P \equiv A_{xy} = (x \rightarrow 0 \land y \rightarrow 0)$ has two distinct minimal upper approximations in $\bar{A}$ namely $Q_1 = A_{xy}$ ($x \rightarrow 0 \land y \rightarrow 0$) and $Q_2 = A_{xy}$ ($x \rightarrow 0 \land y \rightarrow 0$). Now the choice of the most useful upper approximation of the entry assertion $P$ is program-dependent. For example the best choice is $Q_1$ for the program $\chi \equiv x \times y$. This positive declaration can only be justified by performing the two semantic analyses (1.e., $A_{xy} = A_{xy}(Q_1) = A_{xy}(Q_2)$ since $A_{xy} = A_{xy}(Q_1) = A_{xy}(Q_2)$) and next comparing them. Since these analyses are not related by the ordering $\rightarrow$, the comparison-criterion must be application dependent. For example using $Q_1$ we can prove that $A_{xy} = A_{xy}(Q_1) = A_{xy}(Q_2)$ whereas this is impossible with $Q_2$. On the contrary the best choice is $Q_2$ for the program $\chi \equiv x \times y$. Hence $A_{xy} = A_{xy}(Q_1)$ whereas $A_{xy} = A_{xy}(Q_2)$.

End of Example.

If any program must have an analysis which can be approximated from above using $\bar{A}$, 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 $A \subseteq A$ of approximate assertions must be chosen such that for all $P \in A$ the set $(Q, \rightarrow, P \rightarrow Q)$ of upper approximations of $P$ in $\bar{A}$ has a least element.

THEOREM 5.1.0.3

For all $P \in A$ the set $(Q, \rightarrow, P \rightarrow Q)$ has a least element if and only if $\bar{A}$ is a Moore family (i.e., $\bar{A}$ contains the superset of $A$ and is closed under conjunction).

5.2 The Approximation Operator

DEFINITION 5.2.0.1 Approximation Operator

$\rho : A \rightarrow \bar{A}$, $\rho = \lambda P. \bar{A} (Q, \rightarrow, P \rightarrow Q)$

$\rho(P)$ is the least upper approximation of $P$ in $\bar{A}$. Since $\bar{A}$ is a Moore family it follows from Montiero & Ribbe (1972, Th.5.3 and 5.1) that:

THEOREM 5.2.0.2

$\rho$ is an upper closure operator (that is $\rho$ is isotone (if $P \leq Q$ then $\rho(P) \leq \rho(Q)$), extensive (for all $P, Q \in A$, $\rho(P) \leq \rho(Q)$) and idempotent (if $P \leq \rho(P)$).

$\rho(A) = \bar{A}$

$\rho$ is the unique upper closure operator on $A$ such that $\rho(A) = \bar{A}$.

Since $\bar{A}$ is equal to the image of the complete lattice $\bar{A} = \bar{A}(S, \leq, A, \rho)$ by the upper closure operator $\rho$ we derive from Ward (1972, Th.4) the following:

THEOREM 5.2.0.3

$\bar{A}$ is a complete lattice (i.e., $\rho(A)$ is a complete lattice).

$\rho$ is a quasi-complete join-morphism (i.e., $\rho(\psi \setminus A, \rho(VS) \leq \rho(VS))$).

$\bar{A}$ is a complete sub-lattice of $\bar{A}$ if $\rho$ is a complete join-morphism (i.e., $\psi \subseteq A, \rho(VS) \leq \rho(VS)$).
If the initial choice of \( \overline{A} \) does not satisfy assumption 5.1.0.2 we can use the following:

**Theorem 5.2.0.4**

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

\[
\rho(A) = \lambda e : e \subseteq (\overline{A} \cup \{\lambda x. \text{true}\}) \wedge S e
\]

**Example 5.2.0.5**

Returning to example 5.1.0.1 where \( A = (Z \rightarrow B) \) and \( \overline{A} = \{\lambda u. \text{false}, \lambda u. u = 0, \lambda u. u = 0, \lambda u. \text{true}\} \) the least

\( \lambda s. r(v s) = \lambda \) and \( \alpha = \gamma^{-1} \circ \rho \) then

- \( \langle \alpha, \gamma \rangle \) is a pair of adjoined functions
- \( \alpha \) is onto, \( \gamma \) is one-to-one

Reciprocally the approximation process can be defined by the lattice \( A(\varepsilon, 1, T, U, \Pi) \) and a pair of adjoined 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, \( \alpha \in (L_1 \rightarrow L_2) \), \( \gamma \in (L_2 \rightarrow L_1) \). The pair \( \langle \alpha, \gamma \rangle \) defines a Galois connection between \( L_1 \) and \( L_2 \) if and only if:

\[
\begin{align*}
\alpha \circ \gamma &= \text{id} \quad \text{and} \\
\gamma \circ \alpha &= \text{id}
\end{align*}
\]
operator. Moore families can be characterized using definition 5.1.0.2 or theorems 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 \( L(\mathbb{E},1,T,L,\gamma,\eta) \) be a complete lattice and \( f \in L(\mathbb{L}) \).

1. Let \( \alpha \in \{L(\mathbb{L}) \to \mathbb{L}(\mathbb{L}) \} \) be \( \lambda y \gamma(\{f(y) : y \in \mathbb{Y}\}) \).
2. Let \( \varepsilon \in \{L(\mathbb{L}) \to \mathbb{L}(\mathbb{L}) \} \) be \( \lambda y \gamma(\{f(y) : y \in \mathbb{Y}\}) \).
3. Let \( \varepsilon \in \{L(\mathbb{L}) \to \mathbb{L}(\mathbb{L}) \} \) be \( \lambda y \gamma(\{f(y) : y \in \mathbb{Y}\}) \).

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

The composition of two upper closure operators on \( \mathcal{A} \) is usually not a closure operator \([\mathcal{R}(\mathcal{A})]\). 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 \( L(\mathbb{E},1,T,L,\gamma,\eta) \) be a complete lattice, \( P \) an upper closure operator on \( L \) and \( \eta \) be an upper closure operator on \( P(L) \). Then \( \eta \circ \eta \) is an upper closure operator on \( L \) and \( P \in \mathcal{R}(L) \).

Example 6.2.0.2

Many program analysis frameworks are designed in order to describe some properties of each program variable but so that the relationships among the values of these variables are ignored. An example is Jones & Muchnick[75]’s type determination scheme, a counter-example is the determination of linear relationships among numerical variables, Cousot & Halbwachs[75]. The corresponding approximation can be characterized as follows:

Assume that \( V=\mathcal{D}^m \), let \( A \) be \( (V+B) \) and \( A^1 \) be \( (D-B) \). Let us define:

\[
\forall j \in [1,m], \quad \sigma_j \in \mathcal{P}(A) \quad \text{with} \quad \sigma_j = \lambda x \in \mathcal{D}^m \{[\lambda x \in \mathcal{D}^m \{x \in \mathcal{D}^m \}] \}
\]

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.
σ is an upper closure operator on \( A_m \) and an assertion \( P \in A_m \) does not state relationships among the program variables if and only if \( σ(p) = p \). The approximate assertions on each individual program variable \( x_j \) are next defined using an upper closure operator \( ρ_j \) on \( A_1 \). The induced closure operation \( ρ \) on \( A_m \) is defined by \( ρ(P) = \lambda(x_1, \ldots, x_m) = \bigwedge_{j=1}^n ρ_j(P_j)(x_j) \) where \( P \in A_m \) is (necessarily) of the form \( P = \lambda(x_1, \ldots, x_m) \). It follows from theorem 5.2.0.1 that the composition:

\[ ρσ = \lambda P ∈ A_m, \lambda(x_1, \ldots, x_m) ∈ D_m \left( \bigwedge_{j=1}^n ρ_j(σ_j(P))(x_j) \right) \]

is an upper closure operator on \( A_m \).
7. Design of the Approximate Predicate Transformer Induced by a Space of Approximate Assertions

In addition to \( A \) and \( \gamma \) the specification of a program analysis framework also includes the choice of an approximate predicate transformer \( t_c(L \rightarrow A \rightarrow A) \) (or a monoid of maps on \( A \) plus a rule for associating maps to program statements (e.g., Rosen[76])). We now show that in fact this is not indispensable since there exists a best correct choice of \( t \) which is induced by \( A \) and the formal semantics of the considered programming language.

7.1 A Reasonable Definition of Correct Approximate Predicate Transformers

At paragraph 3, given \( \langle V, A, \tau \rangle \) the minimal assertion which is invariant at point \( i \) of a program \( P \) with entry specification \( \Phi \) was defined as:

\[
P_i = \bigvee_{p \in \text{Path}(i)} \tau(p)(\Phi)
\]

Therefore the minimal approximate invariant assertion \( \rho(P_i) \) is the least upper approximation of \( P_i \) in \( \bar{A} \) that is:

\[
\rho(P_i) = \bigvee_{p \in \text{Path}(i)} \tau(p)(\Phi)
\]

Even when \( \text{Path}(i) \) is a finite set of finite paths the evaluation of \( \tau(p)(\Phi) \) is hardly machine-implementable since for each path \( p = a_1, \ldots, a_m \) the computation sequence \( X_0 = \Phi, X_1 = \tau(C(a_1))(X_0), \ldots, X_m = \tau(C(a_m))(X_{m-1}) \) does not necessarily only involve elements of \( \bar{A} \) and \( \langle A, \tau \rangle \). Therefore using \( \Phi \) and \( t_c(L \rightarrow (A \rightarrow A)) \) a machine representable sequence \( X_0 = \Phi, X_1 = \tau(C(a_1))(X_0), \ldots, X_m = \tau(C(a_m))(X_{m-1}) \) is used instead of \( X_0, \ldots, X_m \).
THEOREM 7.1.0.2

(1) - \( \overline{t} \in (L (A + A)) \) is a correct upper approximation of \( t \in (L (A + A)) \) in \( A \) iff \( \forall \{ \text{SL}, \text{WP} \in A, \sigma (S) \Rightarrow \overline{t} (S) \} \)

(2) \( \overline{t} \in (L (A + A)) \) is a correct upper approximation of \( t \in (L (A + A)) \) in \( A \) (where \( A = \{ a, b \} \)) iff \( \forall \{ \text{SL}, \text{WP} \in A, \sigma (S) (y) \Rightarrow \overline{t} (S) (y) \} \).

If \( \overline{t} \in (L (A + A)) \) is any correct upper approximation of \( t \in (L (A + A)) \) in \( A \) then \( \forall \{ \text{SL}, \text{WP} \in A, \sigma (S) \Rightarrow \overline{t} (S) \} \).

Theorem 7.1.0.3

(1) - \( \overline{t} \) is a correct upper approximation of \( t \) in \( A \) iff \( \forall \{ \text{SL}, \text{WP} \in A, \sigma (S) (P) \Rightarrow \overline{t} (S) (P) \} \).

(2) \( \overline{t} \) is a correct upper approximation of \( t \) in \( A \) where \( A = \{ a, b \} \) and \( \sigma (S) \) is isomorphic to \( t \) (where \( \forall \{ \text{SL}, \text{WP} \in A, \sigma (S) \Rightarrow \overline{t} (S) \} \).

Similar results hold for fixpoint analysis of programs.

THEOREM 7.1.0.4

Let \( t \in (L (A + A)) \) be an isomorphic upper approximation of \( t \) in \( A \) (where \( A = \{ a, b \} \) then \( \forall \{ \text{SL}, \text{WP} \in A, \sigma (S) \Rightarrow \overline{t} (S) \} \).

(1) \( F (t, \overline{t}) \) is isomorphic to \( \overline{t} \) and \( \overline{t} (\overline{t}) = \overline{t} (t) \).

(2) \( \overline{t} = \sigma (F (t, \overline{t})) \).

(3) \( \overline{t} = \sigma (F (t, \overline{t})) = \overline{t} (t) \).

Notice that in theorem 7.1.0.2 the maps \( \overline{t} : \text{SL} \) are not assumed to be isomorphic. Yet isomorphism is assumed in theorem 7.1.0.4 and is a customary hypothesis in the literature. An essential justification of this assumption is to ensure that the system of equations \( x = F (t, \overline{t}) (x) \) associated with a program \( t \) has fixpoints which can be obtained as limits of iteration sequences. This can be also achieved without isomorphism taking \( \lambda x. [x \Rightarrow F (t, \overline{t}) (x)] \) instead of \( F (t, \overline{t}) \).

THEOREM 7.1.0.5

Let \( t \in (L (A + A)) \) be a correct upper approximation of \( t \) in \( A \) (where \( A = \{ a, b \} \) then \( \forall \{ \lambda x. \overline{t} (x) \Rightarrow \overline{t} (x) \} \).

(1) \( \lambda x. \overline{t} (x) \Rightarrow \overline{t} (x) \).

(2) \( \lambda x. [x \Rightarrow F (t, \overline{t}) (x)] \).

(3) \( \lambda x. [x \Rightarrow F (t, \overline{t}) (x)] \).

Hence the isomorphism hypothesis is even not necessary for technical purposes. However the profound justification of this hypothesis can be found in the fact that among all possible approximative predicate transformers which can be used with a given set \( A \) of approximate assertions the designer of a program analysis framework intuitively thinks to the best approximate predicate transformer which happens to be isometric. This property also explains the fact that no significant counter-examples to the isometry hypothesis have ever been found.

7.2 The Best Approximate Predicate Transformer Induced by a Space of Approximate Assertions

DEFINITION 7.2.0.1

If \( t_1, t_2 \) are correct upper approximations of \( t \) in \( A \) in \( A \) then we say that \( t_1 \) is better than \( t_2 \) if for all \( \forall \{ \text{SL}, \text{WP} \in A, \sigma (S) \Rightarrow \overline{t} (S) \} \).

LEMMA 7.2.0.2

Let \( t \) be a correct upper approximation of \( t \) in \( A \) in \( A \). If \( \forall \{ \text{SL}, \text{WP} \in A, \sigma (S) \Rightarrow \overline{t} (S) \} \) and \( \overline{t} \) is isometric then \( t_1 \) is better than \( t_2 \). (Notice that the above isometry condition is sufficient but not necessary).

THEOREM 7.2.0.3

Let \( \overline{t} \) be \( \lambda x. \overline{t} (x) = \overline{t} (x) \).

COROLLARY 7.2.0.4

If \( \overline{t} \) is a correct upper approximation of \( t \) in \( A \) then \( \overline{t} = \lambda x. \overline{t} (x) \).

Example 7.2.0.5

Coming back to examples 6.2.0.2 and 6.3.0.5 assume that \( \overline{t} \) is the set of integers included between two bounds \( \overline{t} \). Let \( \overline{t} \) be the complete lattice of \( \overline{t} \) with \( \overline{t} \) and \( \overline{t} \) with \( \overline{t} \). Let \( \overline{t} \) be the supremum of \( \overline{t} \) and \( \overline{t} \) with \( \overline{t} \).

Since \( \overline{t} \) is an injective complete meet-morphism the adjoint function \( \overline{t} : (\overline{t} + \overline{t}) \) is determined by \( \overline{t} = \overline{t} (x) \).

Given \( \overline{t} \) and \( \overline{t} \) let us determine the best correct upper approximation of \( \overline{t} \) in \( A \). Again for lack of space we just study the case of \( \overline{t} (x) \) and \( \overline{t} (y) \).

Since \( \overline{t} \) is the injective complete meet-morphism the adjoint function \( \overline{t} : (\overline{t} + \overline{t}) \) is determined by \( \overline{t} (x) = \overline{t} (x) \).
According to theorem 7.2.0.4 the best upper approximation of $ap(\lambda x,y).(x<y))$ in $A$ is $t = a0ap(\lambda x,y), (x<y))v$. If $P \neq A$ equals $<1,1>$ then $t(P) = <1,1>$ else $P = \{[a,b], [c,d]\}$ where $a \neq b$ and $c \neq d$ in which case $t(P) = d(Q)$ where $Q = \lambda x,y.(a \leq x \land c \leq y \land x < y)$. $d(Q) = \lambda x,y.(f : a \leq x \land c \leq y \land x < y) = \lambda x.(a \leq x \land max(c,x) < d = \lambda x,(a \leq x \land c < d) \exists a \neq d$. The same way $d(Q) = \lambda y, (mz(c,a) \leq y)$. Therefore $t(P) = \{ c \}$ and then $<1,1> \exists a \in [\alpha,\mu] \cap [a,d], [c,d] \cap [a,\alpha]$ proving that this choice in Cousot \& Cousot[77a] was optimal.

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 7.2.0.6.3. Justifying the Data Flow Equations of "Available Expressions"

Let $E$ be the set of expressions. The set $aval(t)$ of expressions which are available at exit of a path $t \in T$ is defined by $aval(t) = \{ g \}$ and $G \in L$.

$aval(t; s) = (aval(t) \cap trim(s(S)) \cup gen(s)$ where $trim(S)$ is the set of expressions in $E$ not killed by the command $S$ while $gen(s)$ is the set of expressions generated by $S$.

An expression is available at some program point $q$ if it is available at exit of every path from the entry point $q$ to $q$. Therefore the set of expressions available at $q$ is $aval(t, \lambda x, \text{true})$ where $aval(t, \lambda x, \text{true}) = \lambda t, \text{true}.aval(t)$. $t \in T$. Since $a$ is a complete join-morphism from $2^T(\alpha, \beta, T, v, n)$ onto $2^E(\alpha, E, \beta, n, v)$, theorem 5.3.0.5. (3,2) defines an adjoint function $Y$. According to theorem 7.2.0.4 the best correct
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:

\[ y \leq p(P)(x, y) \]

solution to program-wide analysis problems since whenever some $t(S)$ is not a complete join-morphism MDP$_n(t, \emptyset)$ can be strictly better than $\lfloor P(F_n(t, \emptyset)) \rfloor$. When $A$ satisfies the ascending chain condition $\lfloor P(F_n(t, \emptyset)) \rfloor$ is computable, which is not necessarily the case of MDP$_n(t, \emptyset)$. In that case a variety of methods can be used (e.g. Rosen[78]) which can find
fixpoint is unnecessarily too complicated. The following construction is preferable:

**LEMMA 9.2.0.2**

Let \( \mathcal{L} \) be a complete lattice and \( \mu \) the least fixpoint.

10. COMBINATION OF PROGRAM ANALYSIS FRAMEWORKS

The ideal method in order to construct a program analyser (to be integrated in optimizing compilers or program verification systems) would consist...
Example 10.1.0.3

\[ \gamma_1(t) = \lambda x, \text{false}, \gamma_1(0) = \lambda x, (x = 0), \gamma_1(t) = \lambda x, (x \neq 0), \gamma_1(1) = \lambda x, (x = 0), \gamma_1(T) = \lambda x, \text{true}, \gamma_2(t) = \lambda x, \text{false}, \gamma_2(0) = \lambda x, (x = 0), \gamma_2(T) = \lambda x, \text{true} \]

Using the ordering \( x \sqsubseteq y \) if \((x = 1) \vee (x = 0) \) and \((x, y_1) \sqsubseteq (x, y_2) \) if \((y_1 = 1) \vee (x = 0, y_2 = 0) \). The meaning of elements of \( \gamma_1 \sqcup \gamma_2 \) can be defined as \( \gamma_1 \cdot \gamma_1 = \gamma_1 \sqcap \gamma_1 \). \( \gamma_2 \cdot \gamma_2 \) is \( \gamma_2 \) if \( x_1 \) \( \gamma_1 \cdot T \) \( x_2 \) \( (x_1 = 0) \) \( (x_2 = 0) \) \( \gamma_1 \cdot T \) \( \gamma_2 \cdot T \). The meaning of \( \gamma_2 \cdot \gamma_2 \) is \( \gamma_2 \) if \( x_1 \) \( \gamma_1 \cdot T \) \( x_2 \) \( (x_1 = 0) \) \( (x_2 = 0) \) \( \gamma_1 \cdot T \) \( \gamma_2 \cdot T \).

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 \( L_1(E_1, T_1, L_1) \) and \( L_2(E_2, T_2, L_2) \) are complete lattices we can define the disjoint sum \( L_1 \sqcup L_2 \) as \( L_1 \sqcup L_2 \sqcup \{ T \} \) with ordering \( x \sqsubseteq y \) if \((x = 1) \vee (y = 1) \) or \((x, y_1) \sqsubseteq (x, y_2) \) if \((y_1 = 1) \vee (x = 0, y_2 = 0) \). The meaning of elements of \( L_1 \sqcup L_2 \) can be defined as \( \gamma_1 \cdot \gamma_1 = \gamma_1 \cdot \gamma_1 \). \( \gamma_2 \cdot \gamma_2 \) is \( \gamma_2 \) if \( x_1 \) \( \gamma_1 \cdot T \) \( x_2 \) \( (x_1 = 0) \) \( (x_2 = 0) \) \( \gamma_1 \cdot T \) \( \gamma_2 \cdot T \). \( \gamma_2 \cdot \gamma_2 \) is \( \gamma_2 \) if \( x_1 \) \( \gamma_1 \cdot T \) \( x_2 \) \( (x_1 = 0) \) \( (x_2 = 0) \) \( \gamma_1 \cdot T \) \( \gamma_2 \cdot T \).

End of Remark.

10.2 Reduced Cardinal Power of Program Analysis Frameworks

The cardinal power \( L^{'}_1 \) with base \( E_2, L_2, T_2 \), \( U_2, L_2 \) and exponent \( L_1(E_1, T_1, L_1) \) is the set of all isomorphisms from \( L_1 \) to \( L_2 \) with \( \neq \). The following program \( [\text{Finn} 74, \text{p.} 179] \) computes \( y \cdot x^y \) for every integer \( x \) and natural number \( y \):

\[ \begin{align*}
&\{ 1 \} \ \text{if} \ y_2 = 0 \ \text{do} \ y_2 = 0 \ \text{end if} \\
&\{ 2 \} \ \text{if} \ \text{odd}(y_2) \ \text{then} \ y_2 = y_2 \cdot y_1 \cdot y_2 \end{align*} \]

The fixpoint analysis with entry condition \( \lambda(y_1, y_2, x_1, x_2), (x_1, x_2) \) using \( A_1 \) leads to the following result for the variable \( y_2 \):

\[ \begin{array}{ccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\hline
y_2 & 1 & T & T & T & T & 0 \\
\end{array} \]

The fixpoint analysis using \( A_2 \) leads to the following result for the variable \( y_2 \):

\[ \begin{array}{ccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\hline
y_2 & T & T & T & T & 0 & 0 & 0 \\
\end{array} \]

According to theorem 10.2.0.1 the direct product of the above analyses cannot yield sharper information. On the other hand using the reduced direct product \( (A_1 \cdot A_2) \) and the corresponding optimal approximate predicate transformer (which takes account of the rule \( \text{if} \ \text{odd} \ \text{then} \ 1 = \text{false} \) we get:

\[ \begin{array}{ccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\hline
y_2 & 1 & T & T & T & T & 0 & 0 & 0 \\
\end{array} \]

End of Example.
using the reduced cardinal product of $A_1$ and $A_2$ yields no information since no relationship can be discovered between $p$ and $X$.

Following theorem 10.2.0.1 we determine that if $g : (A_1 \to A_2)$ then $\gamma(g) = (\gamma_1(t) \land \gamma_2(g(t))) \lor (\gamma_1(g(t)) \land \gamma_2(f))$. Therefore $d(g) = h$ where $h(t_1) = k$, $h(t) = g(t)$, $h(f) = g(f)$, $h(t_1) = g(t) \lor g(f)$. It follows that $d(\lambda x o(A_1 \to A_2))$ is isomorphic to $(t, f) \to A_2$ (or $A_1 \times A_2$).

The system of equations associated with the above program and the entry specification $\lambda b.x$ is then:

\[ g_1 = \lambda b.x \]