Conference Record
of the
FOURTH ACM SYMPOSIUM ON
PRINCIPLES OF PROGRAMMING LANGUAGES

Papers Presented at the Symposium
Los Angeles, California
January 17-19, 1977

Sponsored by the
ASSOCIATION FOR COMPUTING MACHINERY
SPECIAL INTEREST GROUP ON AUTOMATA AND COMPUTABILITY THEORY
SPECIAL INTEREST GROUP ON PROGRAMMING LANGUAGES
1. Introduction

A program denotes computations in some universe of objects. Abstract interpretation of programs consists in using that denotation to describe computations in another universe of abstract objects, so that the results of abstract execution give some informations on the actual computations. An abstract program properties are modeled by a complete semilattice, Birkhoff[61]. Elementary program constructs are locally interpreted by order preserving functions which are used to associate a system of recursive equations with a program. The program global properties are then defined as one of the extreme fixpoints of that system, Tarski[55]. The abstraction process is defined in section 6. It
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 $S^*$ the complete lattice obtained from $S$ by adjoining $\{\bot, \top\}$ to it, and imposing the ordering $\frac{x}{y} < \frac{x}{y'}$ for all $x \in S$.

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

- Environments are used to hold the bindings of identifiers to their values:
  \[
  Env = Idents^\to Values
  \]

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

In particular the projection \( \text{val} \mid \text{Expr} \in \text{domain Bexpr} \) has the functionality:

\[
\text{val} \mid \text{Bexpr} : \text{Bexpr} \to [\text{Env} \to Bool].
\]

- The state set "States" consists of the set of all information configurations that can occur during computations:
  \[\forall s \in States, s = \langle cs(s), env(s) \rangle.\]

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

- If \( e \in Env, v \in Values, x \in Idents \) then \( e[v/x] = k \cdot \text{cond}(v = x, v, e(y)) \)

- The state transition function defines for each state a next state (we consider deterministic programs):

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

\[
\text{n-state}(s) = \begin{cases} 
\text{let } n = \text{end}(cs(s)), e = \text{env}(s) \text{ within case } n \text{ in} \\
\text{Assignments} \mapsto <\text{a-succ}(n), \text{eval} [\text{Expr}(n) \parallel (e)\mid \text{id}(n)> \\
\text{Tests} \mapsto <\text{cond}(\text{eval}(\text{test}(n)) \parallel (e) \mid \text{Bexpr}, \\
<\text{a-succ}(n)\parallel (e), <\text{a-succ}(f)\parallel (e)> \\
\text{Junctions} \mapsto s> > <\text{a-succ}(n), e> \\
\text{Exirs} \mapsto s> > <\text{a-succ}(n), e> \\
\end{cases}
\]

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

- Let \( l_\text{Env} \) be the bottom function on \( Env \) such that \( (\forall e \in Idents, l_\text{Env}(x) = \{Values\}) \).
  Let \( l_\text{states} \) be the subset of initial states:
  \( \text{I-states} = \{<\text{a-succ}(n), l_\text{Env}> \mid m \in \text{Entries}\} \)
A "computation sequence" with initial state $i_s \in I$-states is the sequence:

$$s_n = n \text{-state}^n(i_s) \quad \text{for } n = 0, 1, \ldots$$

where $\text{id}$ is the identity function and $i_{\text{id}}^n = f \circ f^n$.

The initial to final state transition function:

$$n \text{-state}^\infty : \text{States} \to \text{States}$$

is the minimal fixpoint of the functional:

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

Therefore

$$n \text{-state}^\infty = \mu F. (n \text{-state} \circ F)$$

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

$$C_v = F\text{-cont}(C_v)$$

where

$$F\text{-cont} : \text{Context-Vectors} \to \text{Context-Vectors}$$

is defined by:

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

Context-Vectors is a complete lattice with union $\bigcup$ such that $C_v_1 \bigcup C_v_2 = \lambda r. (C_v_1(r) \bigcup C_v_2(r))$.

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

$$(C_v_1 \preceq C_v_2) \iff \forall r \in \text{Arcs}, C_v_1(r) \preceq C_v_2(r)$$
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 = Arcs^0 \rightarrow A-Cont.

Whatever (Cv^0, Cv^0) \in A-Cont \times A-Cont may be, we define:

\[ Cv^0 \sqsupset Cv^0 = \lambda r. Cv^0(r) \circ Cv^0(r) \]

\[ Cv^0 \sqsubseteq Cv^0 = \{ r \in Arcs^0, Cv^0(r) \leq Cv^0(r) \} \]

\[ \sqsupset = \lambda r . \top \] and \[ \sqsubseteq = \lambda r . \bot \]

\langle A-Cont, \sqsupset, \sqsubseteq, \top, \bot \rangle can be shown to be a complete lattice. The function:

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

defines the interpretation of basic instructions. If \( \{ C(q) \} q \in a \cdot \text{pred}(n) \) is the set of input contexts of node n, then the output context on exit arc r of n \( (r \in a \cdot \text{succ}(n)) \) is equal to Int(r, C).

Int is supposed to be order-preserving:

\[ \forall a \in Arcs, \forall (Cv^0, Cv^0) \in A-Cont \times A-Cont \]

\[ \{ Cv^0 \sqsubseteq Cv^0 \} \Rightarrow (\text{Int}(a, Cv^0) \not< \text{Int}(a, Cv^0)) \]

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

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

It is easy to show that Int is order-preserving. Hence it has f-approximations \( \text{Int} \). 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 partially 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 (u) semi-lattice or a meet (n) semi-lattice, thus giving rise to two dual

### 5.3 Examples

#### 5.3.1 Static Semantics of Programs

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

\[ I_{\text{gs}} = < \text{Contexts}, u, \leq, \text{Env}, \emptyset, n \cdot \text{context} > \]

where Contexts, u, \leq, Env, \emptyset, n \cdot \text{context}, Context-Vectors, u, \leq, F-Cont respectively correspond to A-Cont, u, \leq, \top, \bot, \text{Int}, A-Cont, u, \leq, \text{Int}.

#### 5.3.2 Data Flow Analysis

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

"Available expressions" give a classical example. An expression is available on 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 Expr the set of expressions occurring in a program P. Abstract contexts will be sets of available expressions, represented by boolean vectors.
The determination of available expressions, back- 
dominators, intervals, ... requires a forward sys-
tem of equations. Some global flow problems, nota-
biy the live variables and very busy expressions 
require propagating information backward through 
the program graph, they are examples of backward 
systems of equations.

6.5.5 Remarks

Our formal definition of abstract interpretations 
has the completeness property since the model en-
sures 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 \( T = (A-\text{Cont}, \Gamma, \Sigma) \), 
\( \overline{T}, \overline{1}, \overline{\text{Int}} \) of a program is consistent with a "con-
crete" interpretation \( I = (C-\text{Cont}, \rho, \Sigma, \overline{T}, \overline{1}, \overline{\text{Int}}) \) 
if the context vector \( \overline{CV} \) resulting from \( \overline{T} \) is a cor-
rect approximation of the particular context vector \( CV \) 
resulting from the more refined interpretation \( I \). This 
may be rigorously defined by establishing a corre-
spendence (\( \alpha : \text{abstraction} \)) between concrete and ab-
stract context vectors, and inversely (\( \gamma : \text{concreti-
} 

cization} \)), and requiring:

\[
(\gamma(\overline{CV}) \preceq \overline{T} \in (A-\text{Cont}) \) and 
(\( \overline{\alpha}(\overline{CV}) \preceq \overline{C} \))
\]

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

If \( f : D \rightarrow D' \) we note \( \overline{D} = A-\text{Cont}) \) and \( \overline{D'} = A-\text{Cont}) \) 
and \( \overline{f} : \overline{D} \rightarrow \overline{D'} = \lambda x. (\lambda r. f(d(r))) \). 
We will suppose \( \alpha \) and \( \gamma \) to satisfy the following hypothesis:

6.1 \( \alpha : C-\text{Cont} \rightarrow A-\text{Cont}, \gamma : A-\text{Cont} \rightarrow C-\text{Cont} \)

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

6.3 \( \forall \Sigma \in A-\text{Cont}, \Sigma \preceq \gamma(\overline{\Sigma}) \)

6.4 \( \forall \Sigma \in C-\text{Cont}, \Sigma \preceq \gamma(\overline{\Sigma}) \)

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

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

\[
\begin{align*}
(\gamma(\alpha(x)) & \in A-\text{Cont}) \land \\
(\gamma(\alpha(x)) & \in \text{Int}(a, \overline{\gamma(x)}))
\end{align*}
\]

6.5 \( \forall \Sigma \in A-\text{Cont}, \Sigma \preceq \gamma(\overline{\Sigma}) \)

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

\[
\begin{array}{c}
\text{Context} \quad \text{Concrete} \\
\overline{\text{C}} & \overline{\text{C}} _{0} \\
\overline{\gamma} & \overline{\gamma} _{0} \\
\overline{\alpha} & \overline{\alpha} _{0} \\
\end{array}
\]

Suppose we want to compute the concrete output con-
text \( C_0 \) (associated with arc \( a \)) resulting from con-
crete input contexts \( C_1 \) : \( C_0 = \text{Int}(a, C_1) \). We can 
as well approximate this computation in the abstract 
universe, and get \( C'_0 = \gamma(\text{Int}(a, \alpha(C_1))) \). 6.5 requires 
\( C_0 \) to contain at least \( C'_0 \), that is \( C_0 \leq C'_0 \). On 
the contrary we do not require \( C'_0 \) to contain at most 
\( C_0 \), that is \( C'_0 \leq C_0 \) is not compulsory.

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

Note that \( I \leq (a, \gamma)\overline{I} \) imposes a local consistency 
of the interpretations \( I \) and \( \overline{I} \), at the level of pri-
mitive language constructs (6.5). Theorems 7.1 \( \overline{T} \) and 
T2 of Appendix 12 then prove 6.0 which defines the 
global consistency of \( I \) and \( \overline{I} \) at the program level.

In particular if we take

\[
I_{SS} = \langle \text{Contexts}, \rho, \Sigma, \text{Env}, \phi, a-context\rangle
\]

any abstract interpretation \( \overline{I} \) of \( F \), consistent with 
\( I_{SS} \), is consistent with the semantics of \( F \), which implies:

\[
\forall \Sigma \in A-\text{Cont}, \Sigma \preceq \gamma(\overline{\Sigma}) \text{ be the result of } \overline{I},
\]

\[
\forall n \in 0, \exists s \in \text{States} | \langle q, e \rangle = \text{scale}(n)(i_s)
\]

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

Example: Deductive Semantics of Programs

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

\[
I_{DS} = \langle \text{Pred}, \phi, \rightarrow, \text{true}, \text{false}, a-pred \rangle
\]

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

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

\[
\text{let}(n \text{ be origin}
\begin{cases}
\text{Entries} & \Rightarrow (\forall x \in \text{Ident}, x = \text{i}\text{Values}) \\
\text{Junctions} & \Rightarrow \text{or}(P_v(q)) \\
\text{Tests} & \Rightarrow \text{case } n \text{ in} \\
& \Rightarrow (\overline{a\text{-succ}\text{-c}(n)} \Rightarrow P_v(p) \text{ and test}(n)) \\
& \Rightarrow (\overline{a\text{-succ}\text{-c}(n)} \Rightarrow P_v(p) \text{ and not test}(n)) \\
\text{esac} \\
\text{Assignments} & \Rightarrow \text{let}(P \text{ be } P_v(p), (x, e) \text{ be id}((n))\text{ within} \\
& \langle (\forall v \in \text{Values} | \overline{Pv\text{alns}}) \text{ and } x = e\text{aln}(x) \rangle \\
\text{esac} \\
\]

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

The deductive semantics is easily validated by proving that $\Gamma_1 \subseteq (\alpha, \gamma)\Gamma_2$, satisfying:

\[
\alpha : \text{Contexts} \rightarrow \text{Pred} \\
= \forall c \cdot (\gamma (x) \text{ and } x = e(x)) \\
\in \Gamma_1 \otimes \text{Ident} \\
\gamma : \text{Pred} \rightarrow \text{Contexts} \\
= \forall p \cdot (e | P[e(x)/x], x \in \text{Ident}) \\
\]

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

\[
(\forall a \in \text{Arcs}, P_v \in \text{Pred}) \\
\alpha(n\text{-context}(a, \gamma(P_v))) \Rightarrow n\text{-pred}(a, P_v) \\
\]

See Hoare and Lauer[74], Ligle[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 $\Gamma_1 \subseteq (\alpha, \beta)\Gamma_2$ has been proved, we know that the deductive semantics gives a valid proof technique, which will never permit a false theorem to be deduced:

\[
\forall q \in \text{Aracs}, P_v \text{ be the result of } \Gamma_1 \Rightarrow \Gamma_2 \\
\{ n \geq 0, \exists i_{\text{g}} \in \text{I-states} | (q, e) = n\text{-state } \iota_{\text{g}}(i_{\text{g}}) \} \\
\Rightarrow (P_v(q) \Rightarrow \alpha(e)) \\
\]

7. The Lattice of Abstract Interpretations

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

- reflexive: $(I \subseteq (I, I))$ where $= \lambda x.x$ is the identity function,
- transitive: $(I \subseteq (\alpha_1, \gamma_1))$ and $(I' \subseteq (\alpha_2, \gamma_2))$ imply $I \subseteq \alpha_1 \circ \alpha_2, \gamma_2 \circ \gamma_1)$. 

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

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

is an equivalence relation. We have:

\[
\{ I \equiv \beta(I') \} \iff \{ (I \subseteq \beta(I') \text{ and } (\beta^{-1}(I') \subseteq I) \} \\
\]

The proof gives some insight in the abstraction process:

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

2 - reciprocally, if $I \subseteq (\alpha_1, \gamma_1)$, let $\equiv \langle \alpha \rangle$ be the equivalence relation defined on $I$ (properly speaking, on the set of abstract contexts of $I$) by:

\[
\{ x \in (\alpha_1) \equiv \{ x \in (\alpha_1(x) = x \} \} \\
\forall x' \in I', each equivalence class $C_x' = \{ x \in I | (\alpha_1(x) = x \} \} has a least upper bound which is
\gamma_1(x')'. Hence the projection $\alpha_i' \gamma_1(I')' of$ $\alpha_i$ on $\gamma_1'(I')$ is a bijection from the set $\gamma_i(I')$ of representatives of the equivalence classes of $I$.

Let us show now that under the hypothesis $I \subseteq (\alpha_1, \gamma_1)$ and $I' \subseteq (\alpha_2, \gamma_2)$, $\alpha_i$ is bijective on $\gamma_i'(I')$ and $\gamma_2(I')$ is a bijection between $I$ and $I'$. Since $(\alpha_2 \vert \gamma_2(I'))^{-1}$ is a bijection between $I$ and $\gamma_2(I)$, the composition:

\[
(\alpha_1 \vert \gamma_1(\iota_{\text{g}}')) \circ (\alpha_2 \vert \gamma_2(I')) \circ (\alpha_2 \vert \gamma_2(I')^{-1}) \\
= (\alpha_1 \vert \gamma_1(I')) \\
\]

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

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

In particular, we can restrict $I$ to be set of interpretations which abstract $\Gamma_{SS}$ to $I$ is then a lattice, (with ordering $\subseteq$) 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 $\overline{a.S} = [\text{min}(S), \text{max}(S)]$. When $S$ is infinite the bounds will eventually be $-\infty$ and $\overline{(\gamma(a, b))} = \{x | a \leq x \leq b \}$. The abstract contexts are then, (Cousot[76])

\[
\text{Example:} \\
\]
8. Abstract Evaluation of Programs

The system of equations:

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

resulting from an interpretation \( I = \langle A, \text{Cont}, \rightarrow, \leq, \tau, i, \text{Int} \rangle \) 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 of Appendix 12):

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

8.1 Correctness

If \( \text{Int} \) is supposed to be a complete morphism (i.e. infinitely distributive over \( \cdot \)) then \( CV \) is the least fixpoint of \( \text{Int} \), (e.g. Kildall[75]), since in a semi-lattice of finite length, any distributive function is a complete morphism). Under the weaker assumption that \( \text{Int} \) is continuous, the limit \( CV \) of Kleene's sequence can also be shown to be the least fixpoint of \( \text{Int} \) (e.g. Wegbreit[75], since in a well-founded semi-lattice, any isotone function is continuous). Finally, if \( \text{Int} \) is only supposed to be isotone, \( CV \) is an approximation \((\approx)\) of the least fixpoint (e.g. Knu and Ullman[75]).

8.2 Termination

The abstract evaluation terminates iff Kleene's sequence is finite. This may be the case because \( A\text{-Cont} \) is finite (e.g. type checking in ALGOL 60, Nauf[65]), or a finite subset only is to be considered for any particular program (e.g. type checking in ALGOL 68), or \( A\text{-Cont} \) may be of finite length \( m \) (the length of any strictly increasing chain is bounded by \( m \), Kildall[73], Wegbreit[75]) or \( A\text{-Cont} \) 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 (Sintzoff [75]) or heuristics (Katz and Manual[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 $\infty$, with natural ordering $\leq$. The abstract interpretation:

$$I_p = \{ \text{Kir} \}$$

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

$$\text{Kir}(r, C_v) =$$

\[
\begin{align*}
\text{let } n & \text{ be origin}(r) \text{ within } n \text{ in } \text{case } n \text{ in } \text{case } r \text{ in } \text{case } r \text{ in } \text{case } \text{esac} \\
\text{Entries} & \Rightarrow 1 \{ \text{unique entry node} \} \\
\text{Junctions} & \Rightarrow \sum_{e \prec \text{pred}(n)} C_v(p) \\
\text{Assignments} & \Rightarrow \prod_{e \prec \text{pred}(n)} \frac{1}{\text{Prob(test}(n) = \text{true})} \\
\text{Tests} & \Rightarrow \text{Prob(test}(n) = \text{true}) \\
\{a \prec \text{succ}-\text{test}(n)\} & \Rightarrow C_v(\text{pred}(n) + a) \\
\{a \prec \text{succ}-\text{test}(n)\} & \Rightarrow C_v(\text{pred}(n) + a) \\
\text{esac} & \text{esac}
\end{align*}
\]

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

For fixed probabilities, the functionKir is clearly continuous (although it is not a complete morphism) since

$$\text{Kir}( \cdot, C_v) \preceq \ldots \preceq \text{Kir}( \cdot, C_v) \preceq \ldots$$

then

$$\max_{1 \leq p \leq \infty} C_v(\text{pred}(n)) = \sum_{p = 1}^{\infty} \frac{\text{Prob}(\text{test}(n) = \text{true})}{\text{Pred}(\text{test}(n) = \text{true})}$$

and

$$\max_{1 \leq p \leq \infty} (n_1 \cdot q) = \max_{1 \leq p \leq \infty} (n_1) \cdot q.$$ 

The least fixpoint of Kir is the limit of Kleene's sequence (the length of the sequence is in general infinite):

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

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

which is an infinite series. Its sum is $\frac{1}{1-q}$.

This abstract interpretation leads to a system of linear equations. Kleene's sequence corresponds to the Jacobi's iterative method (for numerical coefficients).

9. Fixpoints Approximation Methods

When the extreme fixpoints of the system of equations established for an abstract interpretation $I$ of a program $P$ cannot be computed in finitely many steps, they can be approximated. A more abstract interpretation $\bar{I}$ ($1 \leq \bar{I} \leq 1$) may be used for that purpose (e.g., Ternenbaum[74]). It is often better to make approximations in $\bar{I}$, for example by "accelerating the convergence" of Kleene's sequences.

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

Let $I = \langle A\text{-Cont}, \leq, \preceq, \tau, \text{Int} \rangle$ be an interpretation of $P$. When the least fixpoint $C_v$ of $\text{Int}$ is unreachable, we look for an upper bound $UB$ of $C_v$, since according to the correctness requirement $6.0$, $C_v \preceq \gamma(C_v)$ and $C_v \leq UB$ implies $C_v \leq \gamma(UB)$.

9.1.1 Increasing Approximation Sequence

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

$$\{C \preceq \text{Int}(C) \land \text{not(Int}(C)) \subseteq C\}$$

$$C \preceq \text{Int}(C) \subseteq \text{Int}(C) \preceq C$$

Every finite sequence $I, A\text{-Int}(I), \ldots, A\text{-Int}^k(I), \ldots$ is not strictly increasing. The approximation sequence $S_0, \ldots, S_n, \ldots$ is recursively defined by:

9.1.1.3

$$S_{n+1} = \begin{cases} S_n & \text{if not(Int}(S_n) \nleq S_n) \\ S_0 & \text{else} \end{cases}$$

We now prove that $m$ finite such that:

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

Let $m$ be the least natural (eventually infinite) such that $S_m = S_{m+1}$. We know from 9.1.1.3 that not(Int)(S_k) \preceq S_k. Hence by definition of the ordering $\preceq$, $S_k \nleq \text{Int}(S_k) \preceq S_k$.

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

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

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

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

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

Since $S_m \preceq S_{m+1}$ we have $\text{Int}(S_m) \nleq S_{m+1}$ and therefore $C_v \nleq S_m$. $S_m$ is a correct approximation of $C_v$. 

245
9.1.2 Generalization of Kleene's Ascending Sequence

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

9.1.3 Widening in Increasing Approximation Sequences

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

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

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

As before, we define:

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

Now we have to show that this definition of \( A\text{-int} \) satisfies the requirements 9.1.1.2 and 9.1.1.7.

Let us consider a sequence \( S_0 = 1, \ldots, S_m \) \( = A\text{-int}(S_p) \), \( \ldots \). We show that this sequence is increasing that is to say:

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

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

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

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

If \( q \in \omega\text{-arcs} \), then \( A\text{-int}(q, S_{m+1}) =
\]

\[
\text{Int}(q, Cv/C/r).
\]
Assignment statements are treated using an interval arithmetic (e.g., \([i, j] + [k, l] = [i+k, j+l]\) naturally extended to include the case of the empty interval). Similarly, tests are treated using an "interval logic." Since there exist infinite Kleene's sequences (e.g., \([i, j \downarrow] \in [0, 0] \subset [0, 1] \subset \cdots \subset [0, \infty]\) for the program \(x := 0\); while true do \(x := x + 1\), we must use an approximation sequence. Hence, the results will be somewhat inaccurate but runtime subscript tests may be inserted in the absence of certainty.

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

\[\triangledown = [0, 0] \triangledown [k, l] = \begin{cases} \text{null} & \text{if } k < i \text{ then } -\infty \text{ else } i \in i, \\ \text{null} & \text{if } i > j \text{ then } +\infty \text{ else } j \in j, \\ \end{cases}\]

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

\[(2) \quad C2 \triangleleft C2 \triangledown (C1 \cup C4)\]

The corresponding approximation sequence is:

\[C1 = [1, 1] \quad \text{for } i \in [0, 3]\]
\[C2 \triangleleft C2 \triangledown (C1 \cup C4) = \begin{cases} \text{null} & \text{if } k < i \text{ then } -\infty \text{ else } i \in i, \\ \text{null} & \text{if } i > j \text{ then } +\infty \text{ else } j \in j, \\ \end{cases}\]
\[C3 = C2 \cap [0, 100] = [1, 1] \cap [0, 100] = [1, 1] \]
\[C4 = C3 \cap [1, 100] = [1, 1] \cap [1, 100] = [1, 100] \]
\[C5 = C4 \cap [101, +\infty] = [101, +\infty] \]
\[C6 = [1, +\infty] \cap [101, +\infty] = [101, +\infty] \]
\[C7 = C6 \cap [1, 100] = [1, 100] \]

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

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

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

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

\[S_0 \supseteq S_m \supseteq S_n \supseteq \cdots \supseteq S_0 \supseteq \text{Int}_m^{\infty}(S_m) \supseteq \cdots \supseteq \text{Int}_m(S_m) \supseteq \text{Int}(S_m)\]

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

### 8.3.1 Decreasing Approximation Sequence

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

\[\text{Int}_m^{n+1}(S_m) \supseteq \text{Int}_m^n(S_m) \supseteq \text{Int}(D)\]

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

\[\text{Int}_m^{n+1}(S_m) \supseteq D \supseteq \text{Int}_m(S_m)\]

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

### 8.3.3 Truncated Decreasing Sequence

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

\[D \supseteq 0 \supseteq \text{Int}_m^n(S_m) \supseteq D \supseteq \text{Int}_m^{n+1}(S_m)\]

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

Let \( \text{D-int} : \text{A-Cont} \rightarrow \text{A-Cont} \) be such that:

9.3.2.1 \( \forall C \in \text{A-Cont} \),
\[
(C \subseteq \text{Int}(C)) \implies (C \subseteq \text{D-int}(C) \subseteq \text{Int}(C))
\]

9.3.2.2 \( \forall C \in \text{A-Cont} \), every infinite sequence \( C, \text{D-int}(C), \ldots, \text{D-int}(i)(C), \ldots \) is not strictly decreasing.

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

9.3.2.3 \( S_0' = S_m \)
\[
S_{n+1}' = \begin{cases} S_n' & \text{if } (S_n' \nsubseteq \text{Int}(S_n')) \text{ and } (S_n' \nsubseteq \text{D-int}(S_n')) \\ \text{else} & S_n' \end{cases}
\]

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

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

\[
S_0' - S_m \nsubseteq \text{Int}(S_0') \nsubseteq \text{CV}
\]

If \( p > 0 \) then \( S_0' \nsubseteq \text{Int}(S_0') \), therefore \( S_0' \nsubseteq \text{Int}(S_0') \).

Then applying 9.1.2.1 we have:

\[
S_0' \nsubseteq \text{D-int}(S_0') = S_1' \nsubseteq \text{Int}(S_1') \nsubseteq \text{CV}
\]

But 9.3.2.3 implies \( S_0' \nsubseteq \text{D-int}(S_0') \), hence:

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

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

\[
S_k' \nsubseteq \text{Int}(S_k') \nsubseteq \text{CV}
\]

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

\[
\text{Int}(S_{k+1}') \nsubseteq \text{Int}(S_k') \nsubseteq \text{Int}^2(S_k') \nsubseteq \text{CV}
\]

By transitivity \( S_{k+1}' \nsubseteq \text{Int}(S_k') \) and since 9.3.2.3 implies \( S_k' \nsubseteq \text{Int}(S_k') \) we have from 9.1.2.1:

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

Since 9.3.2.3 implies \( S_k' \nsubseteq \text{D-int}(S_k') \) we have:

\[
S_k' \nsubseteq \text{D-int}(S_k') \nsubseteq \text{CV}
\]

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 Decreasing Sequence

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

The limit of the descending sequence \( S_0' = S_1' = \ldots = S_p' \) is an upper bound of the greatest fixpoint of \( \text{Int} \).

9.3.4 Narrowing in Truncated Decreasing Sequences

By analogy with 9.1.3 we define now the narrowing operation in order to build a possible construction of \( \text{D-int} \) by local modifications to \( \text{Int} \):

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

9.3.4.2 \( \forall (C, C') \in \text{A-Cont}^2 \),
\[
(C \subseteq C') \implies (C \subseteq C \Delta C' \subseteq C')
\]

9.3.4.3 Every infinite sequence \( S_0, S_1, S_2, \ldots \) of the form \( S_n = C_0, S_1 = C_0 \Delta C_1, \ldots, S_n = S_{n-1} \Delta C_n \), \( \ldots \), for arbitrary abstract contexts \( C_0, C_1, \ldots, C_n \), \( \ldots \) is not strictly decreasing.

The approximated interpretation:

\[
\text{D-int} = \text{Arcs} \times \text{A-Cont} \rightarrow \text{A-Cont}
\]

is defined by:

9.3.4.4 \( \text{D-int} = \lambda(q, \text{CV}) \cdot \begin{cases} \text{CV} \nsubseteq \text{Int}(q, \text{CV}) & \text{if } q \in \text{W-arcs} \text{ then} \\ \text{else} & \text{Int}(q, \text{CV}) \end{cases} \)

This definition of \( \text{D-int} \) trivially satisfies the requirement 9.3.4.1 since \( \text{CV} \nsubseteq \text{Int}(\text{CV}) \) implies \( \text{CV}(q) \nsubseteq \text{Int}(q, \text{CV}) \).

9.4 Example: Bounds of Integer Variables

Let us come back to example 9.2. The system of equations was: 

\[
\begin{align*}
(1) \quad C_1 &= [1, 1] \\
(2) \quad C_2 &= C_1 \cup C_4 \\
(3) \quad C_3 &= C_2 \cap [-\infty, 100] \\
(4) \quad C_4 &= C_3 + [1, 0] \\
(5) \quad C_5 &= C_2 \cap [101, \infty]
\end{align*}
\]

The ascending approximation sequence led to the approximate solution:

* \( C_1 = [1, 1] \)
* \( C_2 = [1, \infty] \)
* \( C_3 = [1, 100] \)
* \( C_4 = [2, 101] \)
* \( C_5 = [101, \infty] \)

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

\[
\Delta = \begin{cases} \text{null} & \text{if } i = j \text{ then } k \leq \min(i, k), j \leq \max(i, k) \text{ if } k \text{ else } \infty \text{ then } k \leq \max(i, k), k \leq \min(i, k) \end{cases}
\]

The system of equations was:

\[
\begin{align*}
(1) \quad C_1 &= [1, 1] \\
(2) \quad C_2 &= C_1 \cup C_4 \\
(3) \quad C_3 &= C_2 \cap [-\infty, 100] \\
(4) \quad C_4 &= C_3 + [1, 0] \\
(5) \quad C_5 &= C_2 \cap [101, \infty]
\end{align*}
\]

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

\[
\Delta = \begin{cases} \text{null} & \text{if } i = j \text{ then } k \leq \min(i, k), j \leq \max(i, k) \text{ if } k \text{ else } \infty \text{ then } k \leq \max(i, k), k \leq \min(i, k) \end{cases}
\]

The system of equations was:

\[
\begin{align*}
(1) \quad C_1 &= [1, 1] \\
(2) \quad C_2 &= C_1 \cup C_4 \\
(3) \quad C_3 &= C_2 \cap [-\infty, 100] \\
(4) \quad C_4 &= C_3 + [1, 0] \\
(5) \quad C_5 &= C_2 \cap [101, \infty]
\end{align*}
\]

The system of equations was:

\[
\begin{align*}
(1) \quad C_1 &= [1, 1] \\
(2) \quad C_2 &= C_1 \cup C_4 \\
(3) \quad C_3 &= C_2 \cap [-\infty, 100] \\
(4) \quad C_4 &= C_3 + [1, 0] \\
(5) \quad C_5 &= C_2 \cap [101, \infty]
\end{align*}
\]
Thus narrowing just discards infinite bounds and makes no improvement on finite bounds, it satisfies the requirements of 9.3.4. According to 9.3.4.4 the system of equations is modified by:

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

The descending approximation sequence is:

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

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

9.6 Dual Approximation Methods

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

\(\mathcal{X}\) and \(\mathcal{N}(X)\) non comparable

\[
\begin{align*}
\sim \quad &\mathcal{A}\mathcal{K}\mathcal{S} \\
\mathcal{X} \subseteq \mathcal{N}(X) \\
\mathcal{X} \supseteq \mathcal{N}(X) \\
\mathcal{X} = \mathcal{N}(X) \\
\mathcal{I} \quad &\mathcal{L}\mathcal{F}\mathcal{P} \\
&\mathcal{G}\mathcal{F}\mathcal{P} \\
&\mathcal{D}\mathcal{R}\mathcal{K} \\
\sim \quad &\mathcal{I}
\end{align*}
\]

\(\mathcal{L}\mathcal{F}\mathcal{P}\) and \(\mathcal{G}\mathcal{F}\mathcal{P}\) are the least and greatest fixpoints of \(\mathcal{I}\mathcal{N}\). The ascending (\(\mathcal{A}\mathcal{K}\mathcal{S}\)) and descending (\(\mathcal{D}\mathcal{R}\mathcal{K}\)) Kleene's sequences converge toward \(\mathcal{L}\mathcal{F}\mathcal{P}\) and \(\mathcal{G}\mathcal{F}\mathcal{P}\) respectively. These limits are reached when \(\mathcal{N}\) is continuous. When \(\mathcal{A}\mathcal{K}\mathcal{S}\) is infinite we have proposed to use an ascending approximation sequence (\(\mathcal{A}\mathcal{A}\mathcal{S}\)) to approximate \(\mathcal{L}\mathcal{F}\mathcal{P}\). Its limit may be some fixpoint \(\mathcal{L}\mathcal{F}\mathcal{P}\), or some \(S\), such that \(S \succeq \mathcal{N}(S)\) and \(S \succeq \mathcal{L}\mathcal{F}\mathcal{P}\).

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

The truncated descending sequence TDS is fundamentally different from AAS, since it ensures that the successive approximations starting from \(S_m\) remain in the partition \{\(X\) | \(X \geq \mathcal{N}(X)\}\), so that their limit \(S^*\) is greater than \(\mathcal{L}\mathcal{F}\mathcal{P}\):

\[
\begin{align*}
\mathcal{L}\mathcal{F}\mathcal{P} \\
\mathcal{G}\mathcal{F}\mathcal{P} \\
\mathcal{D}\mathcal{R}\mathcal{K} \\
\mathcal{I}
\end{align*}
\]

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

\[
\begin{align*}
\mathcal{L}\mathcal{F}\mathcal{P} \\
\mathcal{G}\mathcal{F}\mathcal{P} \\
\mathcal{D}\mathcal{A}\mathcal{S} \\
\mathcal{I}
\end{align*}
\]

249
Any of the AAS, TDS, DAS, TAS methods may yield a fixpoint $f_{p}$ which is not the fixpoint $f_{p}$ or $g_{p}$ of interest. None of these methods can improve $f_{p}$ to reach $f_{p}$ or $g_{p}$, therefore a "fixpoint improvement method" is necessary. It is our feeling that such a method could be designed only when considering that A-Gont possesses a richer structure (i.e. for particular applications).

Furthermore, in the AAS, TDS, DAS, TAS sequences the term of rank $n$ is computed only as a function of the term of rank $n-1$, hence these are "separate steps" methods. One can as well imagine to use "bound steps" methods, where the term of rank

**Acknowledgements**

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

**References**


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 back-track programs. Proc. of the third Int. Coll. on Automata, Languages and Programming, Edinburgh, (July).


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


13. Appendix

We note <L, u, ≤, ⊕, ⊖> a complete u-semilattice L, with partial ordering ≤, supremum ⊕ 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 isoton, f is order-preserving or f is monotone to be synonymous and mean:

\[ \forall (x, y) \in L^2, (x \leq y) \Rightarrow \{f(x) \leq f(y)\} \]

\[ \Rightarrow \{f(x, y) \in L^2, (f(x) \land y) \Rightarrow f(x) \lor f(y)\} \]

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

(H1): Let \( \bar{F} \) be an order-preserving function from the complete semi-lattice <L, u, ≤, ⊕, ⊖> in itself.

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

\[ g = \bigvee \{x \mid (x \in L) \land (x \leq f(x))\} \]

\[ f = \bigwedge \{x \mid (x \in L) \land (f(x) \leq x)\} \]

(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 \( \bar{F} \) the greatest and least fixpoints of F.

(H2): Let a and b be such that:

\[ \alpha : L \to \bar{L} \]

\[ \beta : L \to \bar{L} \]

\[ \alpha \text{ is order preserving} \]

\[ \beta \text{ is order preserving} \]

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

(H3.1): (H1), (H2), and \( \forall \{x \in L, \bar{F}(\alpha(x)) \leq \alpha(F(x))\} \]

(H3.2): (H1), (H2), and \( \forall \{x \in L, \gamma(F(x)) \geq F(\gamma(x))\} \]

Proof:
\[ \forall \{x \in L, F(\alpha(x)) \leq \alpha(F(x))\} \]

\[ \forall \{x \in L, \gamma(F(x)) \geq F(\gamma(x))\} \]

\[ \alpha \text{ and } \beta \text{ are order preserving in (H1).} \]

\[ \gamma(\alpha(x)) \leq \alpha(\gamma(x)) \]

\[ \gamma(\beta(x)) \geq \beta(\gamma(x)) \]

\[ \alpha \text{ and } \gamma \text{ transitivity} \]

\[ \alpha(\gamma(x)) \leq \gamma(\alpha(x)) \]

\[ \beta(\gamma(x)) \geq \gamma(\beta(x)) \]

\[ \alpha \text{ and } \beta \text{ are order preserving in (H1).} \]

\[ \forall \{x \in L, \bar{F}(x) \leq x\} \]

\[ \forall \{x \in L, F(x) \geq x\} \]

\[ \alpha \text{ and } \beta \text{ are order preserving in (H1).} \]

\[ \forall \{x \in L, \bar{F}(x) \geq x\} \]

\[ \forall \{x \in L, F(x) \leq x\} \]

\[ \alpha \text{ and } \beta \text{ are order preserving in (H1).} \]

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

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

(L3): Let F : L → L be an order-preserving function from the semi-lattice <L, u, ≤, ⊕, ⊖> in itself, and g and \( \bar{F} \) respectively the least and greatest fixpoints of F, then:

\[ \forall \{x \in L, (g \lor F(x) \geq x) \]
(T1): H1, H1, H2, H3 imply that the greatest fix-points $g$ and $\bar{g}$ of $F$ and $\bar{F}$ are related by:

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

Proof:

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

\[ \bar{g} \leq \bar{\alpha(g)} \leq \bar{\alpha(g)} \quad \text{trivially} \]
\[ \bar{g} \leq \alpha(F(g)) \leq \alpha(g) \quad \text{since } \bar{g} = F(g) \]
\[ \bar{g} \leq \bar{\alpha}(\bar{g}) \leq \alpha(g) \quad \text{H3.1, } \text{isotone, } \leq \text{ transitive} \]
\[ \bar{g} \leq \alpha(g) \quad \text{L3} \]