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

Patrick Cousot* and Radhia Cousot**

Laboratoire d'Informatique, U.S.M.G., BP. 53
38041 Grenoble cedex, France

1. *Introduction*

A program denotes computations in some universe of objects. Abstract interpretation of programs consists in using that denotation to describe computations in another universe of abstract objects, so that the results of abstract execution give some informations on the actual computations. An intuitive example (which we borrow from Sintzoff [72]) is the rule of signs. The text $-1515 * 17$ may be understood to denote computations on the abstract universe $\{(+), (-), (\pm)\}$ where the semantics of arithmetic operators is defined by the rule of signs. The abstract execution $-1515 * 17 \Longrightarrow -(+) * (+) \Longrightarrow (-) * (+) \Longrightarrow (-)$, proves that $-1515 * 17$ is a negative number. Abstract interpretation is concerned by a particular underlying structure of the usual universe of computations (the sign, in our example). It gives a summary of some facets of the actual executions of a program. In general this summary is simple to obtain but inaccurate (e.g. $-1515 + 17 \Longrightarrow -(+) + (+) \Longrightarrow (-) + (+) \Longrightarrow (\pm)$). Despite its fundamentally incomplete results abstract interpretation allows the programmer or the compiler to answer questions which do not need full knowledge of program executions or which tolerate an imprecise answer, (e.g. partial correctness proofs of programs ignoring the termination problems, type checking, program optimizations which are not carried in the absence of certainty about their feasibility, ...).

2. *Summary*

Section 3 describes the syntax and mathematical semantics of a simple flowchart language, Scott and Strachey[71]. This mathematical semantics is used in section 4 to built a more abstract model of the semantics of programs, in that it ignores the sequencing of control flow. This model is taken to be the most concrete of the abstract interpretations of programs. Section 5 gives the formal definition of the abstract interpretations of a program. Abstract program properties are modeled by a complete semilattice, Birkhoff[61]. Elementary program constructs are locally interpreted by order preserving functions which are used to associate a system of recursive equations with a program. The program global properties are then defined as one of the extreme fixpoints of that system, Tarski[55]. The abstraction process is defined in section 6. It is shown that the program properties obtained by an abstract interpretation of a program are consistent with those obtained by a more refined interpretation of that program. In particular, an abstract interpretation may be shown to be consistent with the formal semantics of the language. Levels of abstraction are formalized by showing that consistent abstract interpretations form a lattice (section 7). Section 8 gives a constructive definition of abstract properties of programs based on constructive definitions of fixpoints. It shows that various classical algorithms such as Kildall [73], Wegbreit[75] compute program properties as limits of finite Kleene[52]'s sequences. Section 9 introduces finite fixpoint approximation methods to be used when Kleene's sequences are infinite, Cousot[76]. They are shown to be consistent with the abstraction process. Practical examples illustrate the various sections. The conclusion points out that abstract interpretation of programs is a unified approach to apparently unrelated program analysis techniques.

3. *Syntax and Semantics of Programs*

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

3.1 *Syntax of a Program*

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

$$\underline{\text{n-succ}}, \underline{\text{n-pred}} : \text{Nodes} \rightarrow 2^{\text{Nodes}} \mid (m \in \underline{\text{n-succ}}(n)) \Longleftrightarrow (n \in \underline{\text{n-pred}}(m))$$

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

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

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

* Attaché de Recherche au C.N.R.S., Laboratoire Associé n° 7.

** This work was supported by IRIA-SESORI under grants 75-035 and 76-160.

- An assignment node ($n \in$ Assignments) has one predecessor and one successor (($|$n-pred(n)$|$ = 1) and ($|$n-succ(n)$|$ = 1)). Let "Ident" and "Expr" be the distinct syntactic categories of identifiers and expressions. An assignment node n assigns the value of the right hand-side expression expr(n) to the left hand-side identifier id(n) :

  expr : Assignments $\rightarrow$ Expr

  id : Assignments $\rightarrow$ Ident

- A test node ($n \in$ Tests) has a predecessor and two successors,(($|$n-pred(n)$|$ = 1) and ($|$n-succ(n)$|$ = 2)). The true and false successor nodes are respectively denoted n-succ-t(n) and n-succ-f(n):

  n-succ-t, n-succ-f : Tests $\rightarrow$ Nodes $|$
  ($\forall n \in$ Tests, n-succ(n) = {n-succ-t(n), n-succ-f(n)}).

  Let "Bexpr" be the syntactic category of boolean expressions, each test node n contains a boolean expression test(n) :

  test = Tests $\rightarrow$ Bexpr

- A junction node ($n \in$ Junctions) has one successor and more than one predecessor, (($|$n-succ(n)$|$ = 1) and ($|$n-pred(n)$|$ > 1)). Immediate predecessor nodes of a junction node are not junction nodes, ($\forall n \in$ Junctions, $\forall m \in$ n-pred(n), not(m $\in$ Junctions)).

- An exit node n has one predecessor and no successor, (($|$n-pred(n)$|$ = 1) and (n-succ(n) = $\emptyset$)).

The set "Arcs" of edges of a program is a subset of Nodes $\times$ Nodes defined by :

Arcs = {<n,m> $|$ ($n \in$ Nodes) and ($m \in$ n-succ(n))}

which may be equivalently defined by :

Arcs = {<n,m> $|$ ($m \in$ Nodes) and ($n \in$ n-pred(m))}.

We will assume that the directed graph <Nodes, Arcs> is connected.

We will use the following functions :

origin, end : Arcs $\rightarrow$ Nodes $|$ ($\forall a \in$ Arcs, a = <origin(a), end(a)>)

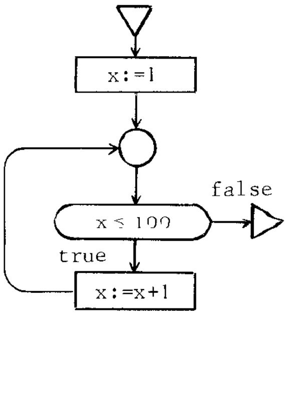
a-succ : Nodes $\rightarrow 2^{Arcs}$ $|$
a-succ(n) = {<n,m> $|$ $m \in$ n-succ(n)}

a-pred : Nodes $\rightarrow 2^{Arcs}$ $|$
a-pred(n) = {<m,n> $|$ $m \in$ n-pred(n)}

a-succ-t : Tests $\rightarrow$ Arcs
a-succ-t(n) = <n, n-succ-t(n)>

a-succ-f : Tests $\rightarrow$ Arcs
a-succ-f(n) = <n, n-succ-f(n)>

Example :



3.2 Semantics of Programs

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

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

- The semantic domain "Values" is a complete lattice which is the sum of the lattice Bool = $\{$true, false$\}^0$ and some other primitive domains.

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

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

  In particular the projection val $|$ Bexpr of the function val in domain Bexpr has the functionality :
  val $|$ Bexpr : Bexpr $\rightarrow$ [Env $\rightarrow$ Bool].

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

  A state ($s \in$ States) consists in a control state (cs(s)) and an environment (env(s)), such that :
  $\forall s \in$ States, s = <cs(s), env(s)>.

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

- If $e \in$ Env, $v \in$ Values, $x \in$ Ident then
  e [v/x] = λy. cond(y = x, v, e(y)).

- The state transition function defines for each state a next state (we consider deterministic programs) :
  n-state : States $\rightarrow$ States

```
n-state(s) =
   let n be end(cs(s)), e be env(s) within
      case n in
         Assignments  ==>
            <a-succ(n),e[val[[expr(n)]] (e)/id(n)]>
         Tests        ==>
            cond(val[[test(n)]] (e) | Bexpr,
              <a-succ-t(n), e>,<a-succ-f(n), e>)
         Junctions    ==> <a-succ(n), e>
         Exits        ==>  s
      esac
```

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

- Let $\bot_{Env}$ be the bottom function on Env such that
  ($\forall x \in \text{Ident}^0$, $\bot_{Env}(x) = \bot_{Values}$).

  Let I-states be the subset of initial states :
  I-states = {<a-succ(m),$\bot_{Env}$> $|$ $m \in$ Entries}

- A "computation sequence" with initial state $i_s \in$ I-states is the sequence :
$$s_n = \underline{\text{n-state}}^n(i_s) \quad \text{for } n = 0, 1, \ldots$$
where f^0 is the identity function and $f^{n+1} = f \circ f^n$.

- The initial to final state transition function :
$$\underline{\text{n-state}}^{\infty} : \text{States} \rightarrow \text{States}$$
is the minimal fixpoint of the functional :
$$\lambda F \,.\, (\underline{\text{n-state}} \circ F)$$
Therefore
$$\underline{\text{n-state}}^{\infty} = Y_{\text{States} \rightarrow \text{States}}(\lambda F \,.\, (\underline{\text{n-state}} \circ F))$$
where $Y_D(f)$ denotes the least fixpoint of $f : D \rightarrow D$, Tarski[55].

4. *Static Semantics of Programs*

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

Hence, we define the context Cq at some program point $q \in$ Arcs of a program P to be the set of all environments which may be associated to q in all the possible computation sequences of P :
$$Cq \in \text{Contexts} = 2^{\text{Env}}$$
$$Cq = \{e \mid (\exists n \geq 0, \exists\, i_s \in \text{I-states} \mid \langle q,e \rangle = \underline{\text{n-state}}^n(i_s))\}$$
The context vector Cv associates a context to each of the program points of a program :
$$\underline{Cv} \in \text{Context-Vectors} = \text{Arcs}^0 \rightarrow \text{Contexts}$$
$$\underline{Cv} = \lambda q \,.\, \{e \mid (\exists n \geq 0, \exists\, i_s \in \text{I-states} \mid \langle q,e \rangle = \underline{\text{n-state}}^n(i_s))\}$$
According to the semantics of programs, the context Cv(r) associated to arc r is related to the contexts Cv(q) at arcs q adjacent to r, ($\underline{\text{end}}(q) = \underline{\text{origin}}(r)$, $\xrightarrow{q}\xrightarrow{r}$). From the definition of the state transition function we can prove the equation :
$$\underline{Cv}(r) = \underline{\text{n-context}}(r, \underline{Cv})$$
where
$$\underline{\text{n-context}} : \text{Arcs}^0 \times \text{Context-Vectors} \rightarrow \text{Contexts}$$
is defined by :

```
n-context(r, Cv) =
   case origin(r) in
      Entries ==>{$\perp_{\text{Env}}$}
      Assignments ∪ Tests ∪ Junctions ==>
         $\bigcup_{q \in \underline{\text{a-pred}}(\underline{\text{origin}}(r))} \; \bigcup_{e \in \underline{Cv}(q)} \underline{\text{env-on}}(r)(\underline{\text{n-state}}(\langle q,e \rangle))$
   esac
```

(We define env-on : $\text{Arcs}^0 \rightarrow [\text{States} \rightarrow 2^{\text{Env}}]$ to be $\lambda r \,.\, (\lambda s \,.\, \underline{\text{cond}}(r = \underline{\text{cs}}(s), \{\underline{\text{env}}(s)\}, \emptyset)))$.

Since the equation Cv(r) = n-context(r, Cv) must be valid for each arc, Cv is a solution to the system of "forward" equations :
$$\underline{Cv} = \underline{\text{F-cont}}(\underline{Cv})$$
where
$$\underline{\text{F-cont}} : \text{Context-Vectors} \rightarrow \text{Context-Vectors}$$
is defined by :
$$\underline{\text{F-cont}}(\underline{Cv}) = \lambda r \,.\, \underline{\text{n-context}}(r, \underline{Cv})$$
Context-Vectors is a complete lattice with union $\tilde{\cup}$ such that $\underline{Cv}_1 \,\tilde{\cup}\, \underline{Cv}_2 = \lambda r \,.\, (\underline{Cv}_1(r) \cup \underline{Cv}_2(r))$.
F-cont is order preserving for the ordering $\tilde{\subseteq}$ of Context-Vectors which is defined by :
$$\{\underline{Cv}_1 \,\tilde{\subseteq}\, \underline{Cv}_2\} \Longleftrightarrow \{\forall r \in \text{Arcs}, \underline{Cv}_1(r) \subseteq \underline{Cv}_2(r)\}$$
Hence it is known that F-cont has fixpoints, Tarski[55]. However, it is trivial to exhibit examples which show that these fixpoints are not always unique. Fortunately, it can be shown that Cv is included in any solution S to the system of equations X = F-cont(X), ($\underline{Cv} \,\tilde{\subseteq}\, \underline{S}$). Tarski[55] shows that this property uniquely determines Cv as the least fixpoint of F-cont. Thus Cv can be equivalently defined by :

$$\text{D1} : \underline{Cv} = \lambda q \,.\, \{e \mid (\exists n \geq 0, \exists\, i_s \in \text{I-states} \mid \langle q,e \rangle = \underline{\text{n-state}}^n(i_s))\}$$
or
$$\text{D2} : \underline{Cv} = Y_{\text{Context-Vectors}}(\underline{\text{F-cont}})$$

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

(α) Cv(q) contains at least the environments e which may be associated to q during any execution of P :
$$\{\exists i \geq 0, \exists\, i_s \in \text{I-states} \mid \langle q,e \rangle \; \underline{\text{n-state}}^i(i_s)\} \Longrightarrow \{e \in \underline{Cv}(q)\}$$

(β) Cv(q) contains only the environments e which may be associated to q during an execution of P :
$$\{e \in \underline{Cv}(q)\} \Longrightarrow \{\exists i \geq 0, \exists\, i_s \in \text{I-states} \mid \langle q,e \rangle = \underline{\text{n-state}}^i(i_s)\}$$

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

5. *Abstract Interpretation of Programs*

5.1 *Formal Definition*

An abstract interpretation I of a program P is a tuple
$$I = \langle \text{A-Cont}, \circ, \leq, \top, \perp, \underline{\text{Int}} \rangle$$
where the set of abstract contexts is a complete $\circ$-semilattice with ordering $\leq$, ($\{x \leq y\} \Longleftrightarrow \{x \circ y = y\}$). This implies that A-Cont has a supremum $\top$. We suppose also A-Cont to have an infimum $\perp$.

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 $\widetilde{\text{A-Cont}} = \text{Arcs}^0 \rightarrow \text{A-Cont}$.
Whatever $(Cv', Cv'') \in \widetilde{\text{A-Cont}}^2$ may be, we define :

$$Cv' \mathbin{\tilde{\circ}} Cv'' = \lambda r\,.\,Cv'(r) \circ Cv''(r)$$

$$Cv' \mathrel{\tilde{\leq}} Cv'' = \{\forall r \in \text{Arcs}^0,\ Cv'(r) \leq Cv''(r)\}$$

$$\tilde{\top} = \lambda r\,.\,\top \text{ and } \tilde{\bot} = \lambda r\,.\,\bot$$

$<\widetilde{\text{A-Cont}}, \tilde{\circ}, \tilde{\leq}, \tilde{\top}, \tilde{\bot}>$ can be shown to be a complete lattice. The function :

$$\underline{\text{Int}} : \text{Arcs}^0 \times \widetilde{\text{A-Cont}} \rightarrow \text{A-Cont}$$

defines the interpretation of basic instructions. If $\{C(q) \mid q \in \underline{\text{a-pred}}(n)\}$ is the set of input contexts of node n, then the output context on exit arc r of n ($r \in \underline{\text{a-succ}}(n)$) is equal to $\underline{\text{Int}}(r, C)$. $\underline{\text{Int}}$ is supposed to be order-preserving :

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

$$\{\underline{Cv'} \mathrel{\tilde{\leq}} \underline{Cv''}\} \Longrightarrow \{\underline{\text{Int}}(a, \underline{Cv'}) \leq \underline{\text{Int}}(a, \underline{Cv''})\}$$

The local interpretation of elementary program constructs which is defined by $\underline{\text{Int}}$ is used to associate a system of equations with the program. We define

$$\underline{\widetilde{\text{Int}}} : \widetilde{\text{A-Cont}} \rightarrow \widetilde{\text{A-Cont}} \mid \underline{\widetilde{\text{Int}}}(\underline{Cv}) = \lambda r\,.\,\underline{\text{Int}}(r, \underline{Cv})$$

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

5.2 Typology of Abstract Interpretations

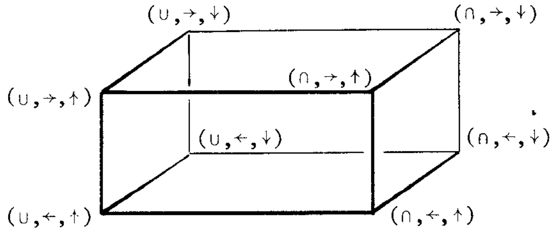
The restriction that "A-Cont" must be a complete semi-lattice is not drastic since Mac Neille[37] showed that any partly ordered set S can be embedded in a complete lattice so that inclusion is preserved, together with all greatest lower bounds and lowest upper bounds existing in S. Hence in practice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

It is a pure coincidence that in most examples (see 5.3.2) the $\cap$ or $\cup$ operator represents the effect of path converging. The real need for this operator is to define completeness which ensures $\underline{\widetilde{\text{Int}}}$ to have extreme fixpoints (see 8.4).

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

Finally we usually consider a maximal ($\uparrow$) or minimal ($\downarrow$) solution to the system of equations, (by agreement, maximal and minimal are related to the ordering $\leq$ defined by $(x \leq y) \Longleftrightarrow (x \cup y = y) \Longleftrightarrow (x \cap y = x)$). However known examples such as Manna and Shamir[75] show that the suitable solution may be somewhere between the extreme ones.

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



Examples :

Kildall[73] uses ($\cap,\rightarrow,\uparrow$), Wegbreit[75] uses ($\cup,\rightarrow,\downarrow$). Tenenbaum[74] uses both ($\cup,\rightarrow,\downarrow$) and ($\cap,\leftarrow,\uparrow$).

5.3 Examples

5.3.1 Static Semantics of Programs

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

$$I_{SS} = <\text{Contexts}, \cup, \subseteq, \text{Env}, \emptyset, \underline{\text{n-context}}>$$

where Contexts, $\cup$, $\subseteq$, Env, $\emptyset$, $\underline{\text{n-context}}$, Context-Vectors, $\tilde{\cup}$, $\tilde{\subseteq}$, $\underline{\text{F-Cont}}$ respectively correspond to A-Cont, $\circ$, $\leq$, $\top$, $\bot$, $\underline{\text{Int}}$, $\widetilde{\text{A-Cont}}$, $\tilde{\circ}$, $\tilde{\leq}$, $\underline{\widetilde{\text{Int}}}$.

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_P be the set of expressions occuring in a program P. Abstract contexts will be sets of available expressions, represented by boolean vectors :

$$\text{B-vect} : \text{Expr}_P \rightarrow \{\underline{\text{true}}, \underline{\text{false}}\}$$

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

```
avail(r, Bv)
   let n be origin(r) within
      case n in
         Entries ⟹ λe . false
         Assignments ∪ Tests ∪ Junctions ⟹
          λe.(generated(n)(e) or ((   and      Bv(p)(e))
                                   p∈a-pred(n)
                               and transparent(n)(e)))
      esac
```

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

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

$$\underline{Bv} = \underline{\widetilde{\text{avail}}}(\underline{Bv})$$

The determination of available expressions, back-dominators, intervals, ... requires a forward system of equations. Some global flow problems, notably the live variables and very busy expressions require propagating information backward through the program graph, they are examples of backward systems of equations.

5.3.3 Remarks

Our formal definition of abstract interpretations has the completeness property since the model ensures 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 reference to the program syntax (e.g. the algorithm for finding available expressions is justified by reasoning 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 $\overline{I} = <\overline{A\text{-}Cont}, \bar{\circ}, \bar{\leq}, \bar{\top}, \bar{\bot}, \overline{Int}>$ of a program is consistent with a "concrete" interpretation $\underline{I} = <C\text{-}Cont, \circ, \leq, \top, \bot, \underline{Int}>$ if the context vector $\overline{Cv}$ resulting from $\overline{I}$ is a correct approximation of the context vector $\underline{Cv}$ resulting from the more refined interpretation $\underline{I}$. This may be rigorously defined by establishing a correspondence ($\tilde{\alpha}$: abstraction) between concrete and abstract context vectors, and inversely ($\tilde{\gamma}$: concretization), and requiring :

6.0 $\{\underline{Cv} \mathrel{\tilde{\leq}} \tilde{\gamma}(\overline{Cv})\}$ $\underline{\text{and}}$ $\{\tilde{\alpha}(\underline{Cv}) \mathrel{\tilde{\leq}} \overline{Cv}\}$

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 $\tilde{D} = Arcs^0 \rightarrow D$ and $\tilde{D}' = Arcs^0 \rightarrow D'$ and $\tilde{f} : \tilde{D} \rightarrow \tilde{D}' = \lambda d . (\lambda r . f(d(r))$.
We will suppose α and γ to satisfy the following hypothesis :

6.1 $\alpha : C\text{-}Cont \rightarrow \overline{A\text{-}Cont}$, $\gamma : \overline{A\text{-}Cont} \rightarrow C\text{-}Cont$

6.2 α and γ are order-preserving

6.3 $\forall \bar{x} \in \overline{A\text{-}Cont}, \bar{x} = \alpha(\gamma(\bar{x}))$

6.4 $\forall x \in C\text{-}Cont, x \leq \gamma(\alpha(x))$

Intuitively, hypothesis 6.2 is necessary because context inclusion (that is property comparison) must be preserved by the abstraction or concretization process. 6.3 requires that concretization introduces no loss of information. It implies that α is surjective and γ 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 $\gamma(\alpha(C)) \geq C$. Note that it is easy to prove properties corresponding to 6.1 - 6.4 for $\tilde{\alpha}$ and $\tilde{\gamma}$.

Instead of the global hypothesis 6.0 we will use the following local hypothesis on the concrete and abstract interpretations of primitive language constructs :

$$\{\forall (a, \bar{x}) \in Arcs \times \widetilde{\overline{A\text{-}Cont}}, \quad \gamma(\overline{Int}(a, \bar{x}) \geq \underline{Int}(a, \tilde{\gamma}(\bar{x}))\}$$

6.5 $\underline{\text{and}}$

$$\{\forall (a, x) \in Arcs \times \widetilde{C\text{-}Cont}, \quad \overline{Int}(a, \tilde{\alpha}(x)) \mathrel{\bar{\geq}} \alpha(\underline{Int}(a, x))\}$$

These two hypothesis are in fact equivalent (lemma L2 in appendix 12). The following schema illustrates 6.5, i.e. the idea of abstract simulation of concrete computations :

$$\begin{tikzcd}
\tilde{\overline{C}}_I \arrow[r, "\overline{Int}(a, \tilde{\overline{C}}_I)"] & \overline{C}_0 \arrow[d, dashed, "\gamma"] \\
 & C'_0 \arrow[d, phantom, "\rotatebox{-90}{$\geq$}"] \\
\tilde{C}_I \arrow[uu, dashed, "\tilde{\alpha}"] \arrow[r, "\underline{Int}(a, \tilde{C}_I)"'] & C_0
\end{tikzcd}$$

Suppose we want to compute the concrete output context C_0 (associated with arc a) resulting from concrete input contexts $\tilde{C}_I$: $C_0 = \underline{Int}(a, \tilde{C}_I)$. We can as well approximate this computation in the abstract universe, and get $C'_0 = \gamma(\overline{Int}(a, \tilde{\alpha}(\tilde{C}_I)))$. 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 $\overline{I}$ is an abstraction of I, denoted $I \leq (\alpha, \gamma)\overline{I}$, if and only if there exist α and γ satisfying hypothesis 6.1 to 6.5.
Note that $I \leq (\alpha, \gamma)\overline{I}$ imposes a local consistency of the interpretations I and $\overline{I}$, at the level of primitive language constructs (6.5). Theorems T1 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

$$I_{SS} = <Contexts, \cup, \underline{\subset}, Env, \emptyset, \underline{n\text{-}context}>$$

any abstract interpretation $\overline{I}$ of P, consistent with I_{SS} ($I_{SS} \leq (\alpha, \gamma)\overline{I}$) is consistent with the semantics of P, which implies :

$\forall q \in Arcs$, let $\overline{Cv}$ be the result of $\overline{I}$,

$$\{\exists n \in 0, \exists i_s \in I\text{-}States \mid <q,e> = \underline{n\text{-}state}^n(i_s)\} \Longrightarrow \{e \in \gamma(\overline{Cv}(q))\}$$

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

Example : Deductive Semantics of Programs

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

$$I_{DS} = <Pred, \underline{or}, \Longrightarrow, \underline{true}, \underline{false}, \underline{n\text{-}pred}>$$

where n-pred defines Floyd[67]'s strongest post condition :

```
n-pred(r, Pv) =
  let(n be origin(r)), (p be a-pred(origin(r)))within
    case n in
      Entries     ==>(∀x ∈ Ident, x = ⊥_Values)
      Junctions ==>    or       (Pv(q))
                    q∈a-pred(n)
      Tests       ==>case r in
                       {a-succ-t(n)} => Pv(p) and
                                                test(n)
                       {a-succ-f(n)} => Pv(p) and
                                           not test(n)
                     esac
      Assignments ==>
          let (P be Pv(p)), (x be id(n)),
                                    (e be expr(n)) within
             (∃v ∈Values | P[v/x] and x = e[v/x])
      esac
```

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

The deductive semantics is easily validated by proving that $I_{SS} \le (\alpha, \gamma) I_{DS}$, where :

$$\alpha : \text{Contexts} \rightarrow \text{Pred}$$
$$= \lambda C . \big(\underset{e \in C}{\text{or}} \big(\underset{x \in \text{Ident}}{\text{and}} (x = e(x)) \big) \big)$$

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

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

$$\{\forall a \in \text{Arcs},\ \forall Pv \in \widetilde{\text{Pred}},$$
$$\alpha(\text{n-context}(a, \tilde{\gamma}(Pv))) \Longrightarrow \text{n-pred}(a, Pv)\}$$

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

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

$$\forall q \in \text{Arcs, let } Pv \text{ be the result of } I_{DS},$$
$$\{\exists n \ge 0,\ \exists\ i_s \in \text{I-states} \mid \langle q,e \rangle = \text{n-state}^{n}(i_s)\}$$
$$\Longrightarrow \{Pv(q) \Longrightarrow \alpha(e)\}$$

7. *The Lattice of Abstract Interpretations*

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

reflexive : $(I \le (\iota, \iota) I)$ where $\iota = \lambda x . x$ is the identify function,

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

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

$$\{I \equiv I'\} \Longleftrightarrow \{(I \le I') \text{ and } (I' \le I)\}$$

is an equivalence relation. We have :

$$\{I \equiv (\beta) I'\} \Longleftrightarrow \{\beta \text{ is an isomorphism between the algebras } I \text{ and } I'\}$$

The proof gives some insight in the abstraction process :

1 - $\{I \equiv (\beta) I'\} \Longrightarrow \{(I \le (\beta, \beta^{-1}) I') \text{ and } (I' \le (\beta^{-1}, \beta) I)\}$

2 - reciprocally,

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

$$\{x \equiv (\alpha_1) y\} \Longleftrightarrow \{\alpha_1(x) = \alpha_1(y)\}$$

$\forall x' \in I'$, each equivalence class $C_{x'} = \{x \in I \mid \alpha_1(x) = x'\}$ has a least upper bound which is $\gamma_1(x')$. Hence the projection $\alpha_1 \mid \gamma_1(I')$ of α_1 on $\gamma_1(I')$ is a bijection from the set $\gamma_1(I')$ of representers of the equivalence classes on I.

Let us show now that under the hypothesis $I \le (\alpha_1, \gamma_1) I'$ and $I' \le (\alpha_2, \gamma_2) I$, α_1 is bijective.

$\alpha_1 \mid \gamma_1(I')$ and $\alpha_2 \mid \gamma_2(I)$ are bijections, hence $\forall x' \in I'$, $\exists! x$ (unique) $\in \gamma_1(I')$ such that $x' = (\alpha_1 \mid \gamma_1(I'))(x)$. Likewise, $x \in \gamma_1(I') \Longrightarrow x \in I \Longrightarrow \exists! x'' \in \gamma_2(I) \mid x = (\alpha_2 | \gamma_2(I))(x'')$. Therefore, $\forall x' \in I'$, $\exists! x'' \in \gamma_2(I) \mid x' = (\alpha_1 \mid \gamma_1(I')) \circ (\alpha_2 \mid \gamma_2(I))(x'')$. Thus $(\alpha_1 \mid \gamma_1(I')) \circ (\alpha_2 \mid \gamma_2(I))$ is a bijection between $\gamma_2(I)$ and I'. Since $(\alpha_2 \mid \gamma_2(I))^{-1}$ is a bijection between I and $\gamma_2(I)$, the composition

$$(\alpha_1 \mid \gamma_1(I')) \circ (\alpha_2 \mid \gamma_2(I)) \circ (\alpha_2 \mid \gamma_2(I))^{-1}$$
$$= (\alpha_1 \mid \gamma_1(I'))$$

is a bijection between I and I', hence α_1 is a bijection between I and I' which is trivially an algebraic morphism. (α_1 is isotone, its inverse $\alpha_1^{-1} = \gamma_1$ is isotone and $\alpha_1(\text{Int}(a, X)) = \text{Int}'(a, \tilde{\alpha}_1(X))$ Q.E.D.

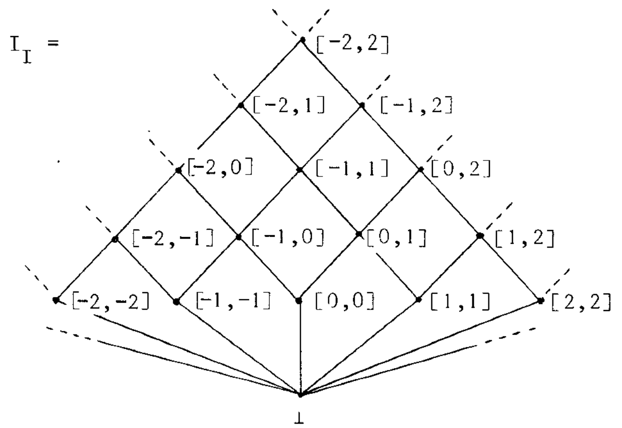
Let *I* be the set of abstract interpretations of a program, if equivalent interpretations are identified, the quasi-ordering $\le$ becomes a partial ordering.

In particular, we can restrict *I* to be set of interpretations which abstract I_{SS}. *I* is then a lattice, (with ordering $\le$) which is isomorphic with a subset of the lattice of equivalence relations on Contexts.

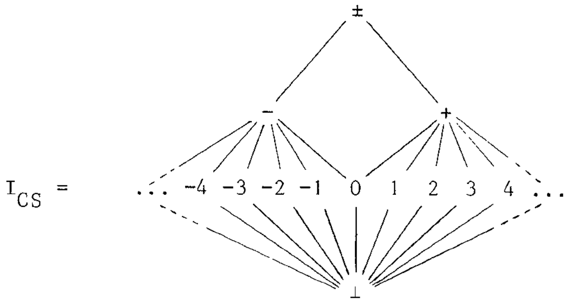
Example :

Let P be a program with a single integer variable, (the generalization is obvious). Environments will be integers (the value of the variable). Contexts are sets of integers (the set of values at some program point).

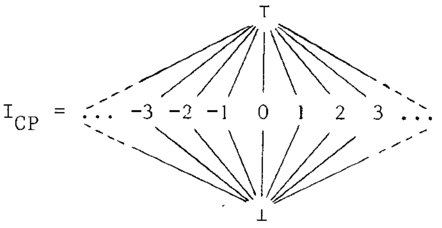
A context S may be abstracted by a closed interval $\alpha(S) = [\text{min}(S), \text{max}(S)]$. When S is infinite the bounds will eventually be $-\infty$ or $+\infty$.
$\gamma([a, b]) = \{x \mid a \le x \le b\}$. The abstract contexts are then, (Cousot[76]) :

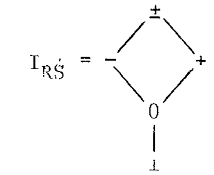


A further abstraction may be :
$\alpha([a, b])$ = if a = b then a elsif a ≥ 0 then + elsif b ≤ 0 then − else ± fi. $\gamma(n) = [n, n]$, $\gamma(+) = [0, +\infty]$, $\gamma(-) = [-\infty, 0]$, $\gamma(\pm) = [-\infty, +\infty]$. The abstract contexts are then :

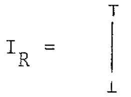


This interpretation may be abstracted by two non-comparable abstractions :





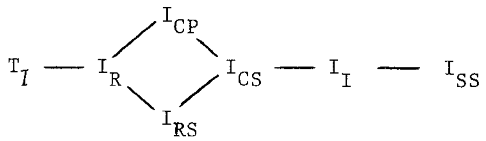
I_{CP} is used by Kildall[73] for constant propagation. I_{RS} might be used to apply the rules of signs. Both interpretations may be abstracted by :



which may be used to check that any vertex in the program graph is reachable from the entry nodes. Finally, the most abstract interpretation is the upper bound of $\mathcal{I}$:

$T_{\mathcal{I}}$ = <{I}, λ(x, y) . I, t, I, I, λ(a, C) . I>

where t is the relation which is always true. We have exhibited a sublattice of $\mathcal{I}$ which is :



8. *Abstract Evaluation of Programs*

The system of equations :

$$\underline{Cv} : \widetilde{\underline{Int}}(\underline{Cv})$$

resulting from an interpretation I = <A-Cont, ∘, ≤ , T, ⊥, Int> 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) :

$\underline{Cv}$:= (C := $\tilde{\bot}$; until C = $\widetilde{\underline{Int}}$(C) do C := $\widetilde{\underline{Int}}$(C) repeat;C)

8.1 *Correctness*

If Int is supposed to be a complete morphism (i.e. infinitely distributive over ∘) then Cv is the least fixpoint of $\widetilde{\underline{Int}}$. (e.g. Kildall[73], since in a semi-lattice of finite length, any distributive function is a complete morphism). Under the weaker assumption that Int is continuous, the limit Cv of Kleene's sequence can also be shown to be the least fixpoint of $\widetilde{\underline{Int}}$ (e.g. Wegbreit[75], since in a well-founded semi-lattice, any isotone function is continuous). Finally, if Int is only supposed to be isotone, Cv is an approximation ($\tilde{\leq}$) of the least fixpoint (e.g. Kam and Ullman[75]).

8.2 *Termination*

The abstract evaluation terminates iff Kleene's sequence is finite. This may be the case because A-Cont is finite (e.g. type checking in ALGOL 60, Naur[65]), or a finite subset only is to be considered for any particular program (e.g. type checking in ALGOL 68), or A-Cont may be of finite length m (the length of any strictly increasing chain is bounded by m, Kildall[73], Wegbreit[75]) or A-Cont may satisfy the ascending chain condition (every strictly increasing chain is finite, although not bounded). A lattice may have infinite chains, although 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 Manna[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 ∞, with natural ordering $\leq$. The abstract interpretation :

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

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

```
Kir(r, Cv) =
  let n be origin(r) within
    case n in
      Entries ==> 1  {unique entry node}
      Junctions ∪ Assignments ==> $\sum_{p \in \text{a-pred}(n)} \text{Cv}(p)$
      Tests    ==>
        case r in
          {a-succ-t(n)} ==> Cv(a-pred(n)) *
                               Prob(test(n) = true)
          {a-succ-f(n)} ==> Cv(a-pred(n)) *
                               (1-Prob(test(n) = true))
        esac
      esac
```

The main difficulty is to obtain the probability Prob(test(n) = true) of taking the true path at a test node n. Suppose the values of these probabilities can be determined (from hypothesis on the input data).
For fixed probabilities, the function $\widetilde{\text{Kir}}$ is clearly continuous (although it is not a complete morphism) since

if $\underline{\text{Cv}}_0 \mathrel{\tilde{\leq}} \underline{\text{Cv}}_1 \mathrel{\tilde{\leq}} \ldots \mathrel{\tilde{\leq}} \underline{\text{Cv}}_n \mathrel{\tilde{\leq}} \ldots$

then $\underset{i=0}{\overset{\infty}{\underline{\max}}} \left(\sum_{p \in \underline{\text{a-pred}}(n)} \underline{\text{Cv}}_i(p) \right) = \sum_{p \in \underline{\text{a-pred}}(n)} \left(\underset{i=0}{\overset{\infty}{\widetilde{\underline{\max}}}} (\underline{\text{Cv}}_i)(p) \right)$

and $\underset{i \in \Delta}{\underline{\max}} (n_i * q) = (\underset{i \in \Delta}{\underline{\max}} (n_i)) * q.$

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

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

- Let P be the program "begin while T do 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 :

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

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

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

Let I = <A-Cont, $\circ$, $\leq$, $\bot$, $\top$, Int> be an interpretation of P. When the least fixpoint $\overline{\text{Cv}}$ of $\widetilde{\text{Int}}$ is unreachable, we look for an upper bound $\overline{\text{Ub}}$ of $\overline{\text{Cv}}$, since according to the correctness requirement 6.0, $\underline{\text{Cv}} \leq \gamma(\overline{\text{Cv}})$ and $\overline{\text{Cv}} \mathrel{\tilde{\leq}} \overline{\text{Ub}}$ implies $\underline{\text{Cv}} \leq \gamma(\overline{\text{Ub}})$.

9.1.1 Increasing Approximation Sequence

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

9.1.1.1 $\{\forall n \geq 0,\ C = \widetilde{\text{A-int}}^n(\tilde{\bot})$ and $\text{not}(\widetilde{\text{Int}}(C) \mathrel{\tilde{\leq}} C)\}$ $\Longrightarrow \{C \mathrel{\tilde{\circ}} \widetilde{\text{Int}}(C) \mathrel{\tilde{\lneq}} \widetilde{\text{A-int}}(C)\}$.

9.1.1.2 Every infinite sequence $\tilde{\bot}$, $\widetilde{\text{A-int}}(\tilde{\bot})$, ..., $\widetilde{\text{A-int}}^n(\tilde{\bot})$, ... is not strictly increasing.

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

```
9.1.1.3  $S_0 = \tilde{\bot}$
         $S_{n+1}$ = if not($\widetilde{\text{Int}}(S_n) \mathrel{\tilde{\leq}} S_n$) then
                    $\widetilde{\text{A-int}}(S_n)$
                 else
                       $S_n$
                 fi
```

We now prove that $\exists m$ finite such that :

$$S_0 \mathrel{\tilde{<}} S_1 \mathrel{\tilde{<}} \ldots \mathrel{\tilde{<}} S_m = S_{m+1} = \ldots$$

Let m be the least natural number (eventually infinite) such that $S_m = S_{m+1}$. $\forall k \in [0, m[$, we know from 9.1.1.3 that $\text{not}(\widetilde{\text{Int}}(S_k) \mathrel{\tilde{\leq}} S_k)$. Whence by definition of the ordering $\tilde{\leq}$, $S_k \neq \widetilde{\text{Int}}(S_k) \mathrel{\tilde{\circ}} S_k$. Since $S_k \mathrel{\tilde{\leq}} \widetilde{\text{Int}}(S_k) \mathrel{\tilde{\circ}} S_k$ is always true , we can state that $S_k \mathrel{\tilde{<}} \widetilde{\text{Int}}(S_k) \mathrel{\tilde{\circ}} S_k$. Besides $\text{not}(\widetilde{\text{Int}}(S_k) \mathrel{\tilde{\leq}} S_k)$ and 9.1.1.1 imply :

$$S_{k+1} = \widetilde{\text{A-int}}(S_k) \mathrel{\tilde{\geq}} \widetilde{\text{Int}}(S_k) \mathrel{\tilde{\circ}} S_k$$

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

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

$$\forall X \in \widetilde{\text{A-Cont}},\ \{\widetilde{\text{Int}}(X) \mathrel{\tilde{\leq}} X\} \Longrightarrow \{\overline{\text{Cv}} \mathrel{\tilde{\leq}} X\}$$

Since $S_m = S_{m+1}$ we have $\widetilde{\text{Int}}(S_m) \mathrel{\tilde{\leq}} S_m$ and therefore $\overline{\text{Cv}} \mathrel{\tilde{\leq}} S_m$. S_m is a correct approximation of $\overline{\text{Cv}}$.

9.1.2 Generalization of Kleene's Ascending Sequence

When A-Cont satisfies the ascending chain condition one can choose $\underline{\widetilde{\text{A-int}}}$ to be $\underline{\widetilde{\text{Int}}}$ 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 $\underline{\widetilde{\text{A-int}}}$ in 9.1.1 is global. We now indicate a way to construct $\underline{\text{A-int}}$ by local modifications to $\underline{\text{Int}}$.

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

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

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

In any infinite strictly increasing Kleene's sequence $\underline{Cv}_1, \ldots, \underline{Cv}_m, \ldots$ since Arcs is finite there is some arc q for which the sequence $\underline{Cv}_1(q), \ldots, \underline{Cv}_m(q), \ldots$ never stabilizes. Therefore q must belong to a cycle or the contexts associated to q transitively depend on the contexts associated to some other arc r which itself belongs to a cycle. The sequence of contexts associated to any arc of that cycle never stabilizes. In order to avoid this phenomenon, we introduce :

- The binary operation ∇ called widening defined by :

  9.1.3.1 ∇ : A-Cont × A-Cont → A-Cont

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

  9.1.3.3 Every infinite sequence $s_0, \ldots, s_n, \ldots$ of the form $s_0 = C_0, \ldots, s_n = s_{n-1} \nabla C_n, \ldots$ (where $C_0, \ldots, C_n, \ldots$ are arbitrary abstract contexts) is not strictly increasing.

- The set W-arcs of widening arcs, which is one of the minimal sets of arcs such that any cycle $<q_1, \ldots, q_n>$ of the system of equations $\underline{Cv} = \underline{\widetilde{\text{Int}}}(\underline{Cv})$ contains at least a widening arc : $\exists i \in [1, n] \mid q_i \in$ W-arcs. (e.g. in a forward interpretation on a reducible program graph, W-arcs may be chosen to be the set of exit arcs of the junction nodes which are interval headers. On irreducible graphs an arbitrary choice has to be made so that any loop of the program goes through a widening arc).

- The approximate interpretation $\underline{\text{A-int}}$: $\text{Arcs}^0 \times \widetilde{\text{A-Cont}} \to \text{A-Cont}$ defined by :

  9.1.3.4 $\underline{\text{A-int}} = \lambda(q, \underline{Cv})$. $\underline{\text{if}}$ q ∈ W-arcs $\underline{\text{then}}$
  $\underline{Cv}(q)\ \nabla\ \underline{\text{Int}}(q, \underline{Cv})$
  $\underline{\text{else}}$
  $\underline{\text{Int}}(q, \underline{Cv})$
  $\underline{\text{fi}}$

As before, we define :

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

Now we have to show that this definition of $\underline{\widetilde{\text{A-int}}}$ satisfies the requirements 9.1.1.2 and 9.1.1.1.

Let us consider a sequence $S_0 = \tilde{\bot}, \ldots, S_{n+1} = \underline{\widetilde{\text{A-int}}}(S_n), \ldots$ We show that this sequence is increasing that is to say :

9.1.3.6 $S_n \tilde{\leq} \underline{\widetilde{\text{A-int}}}(S_n),\ \forall n \geq 0.$

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

$$S_{m+1} \tilde{\leq} \underline{\widetilde{\text{A-int}}}(S_{m+1})$$

$\Longleftrightarrow\ S_{m+1}(q) \leq \underline{\text{A-int}}(q, S_{m+1}), \forall q \in \text{Arcs}.$

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

If q ∉ W-arcs, then

$$\underline{\text{A-int}}(q, S_{m+1}) = \underline{\text{Int}}(q, S_{m+1})$$

$\Longrightarrow\ \underline{\text{Int}}(q, S_m) \leq \underline{\text{A-int}}(q, S_{m+1})$ since $S_m \tilde{\leq} S_{m+1}$ and $\underline{\text{Int}}$ is order preserving. Moreover from q ∉ W-arcs and 9.1.3.4 we get $\underline{\text{Int}}(q, S_m) = \underline{\text{A-int}}(q, S_m)$ and therefore $S_{m+1}(q) = \underline{\text{A-int}}(q, S_m) \leq \underline{\text{A-int}}(q, S_{m+1})$.

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

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

We now prove 9.1.1.1 that is to say that :

$$\forall n \geq 0,\ S_n = \underline{\widetilde{\text{A-int}}}^n(\tilde{\bot})$$

implies

$$S_n \tilde{\circ} \underline{\widetilde{\text{Int}}}(S_n) \tilde{\leq} \underline{\widetilde{\text{A-int}}}(S_n)$$

$\Longleftrightarrow\ (S_n \tilde{\circ} \underline{\widetilde{\text{Int}}}(S_n))(q) \tilde{\leq} \underline{\widetilde{\text{A-int}}}(S_n)(q),\ \forall q \in \text{Arcs}$

$\Longleftrightarrow\ S_n(q) \circ \underline{\text{Int}}(q, S_n) \leq \underline{\text{A-int}}(q, S_n)$ (see 9.1.3.5)

If q ∈ W-arcs, we have $\underline{\text{A-int}}(q, S_n) = S_n(q)\ \nabla\ \underline{\text{Int}}(q, S_n) \geq S_n(q) \circ \underline{\text{Int}}(q, S_n)$ by 9.1.3.2. If now q ∉ W-arcs we must show :

$$S_n(q) \circ \underline{\text{Int}}(q, S_n) \leq \underline{\text{Int}}(q, S_n)$$

$\Longleftrightarrow\ S_n(q) \circ \underline{\text{Int}}(q, S_n) = \underline{\text{Int}}(q, S_n)$

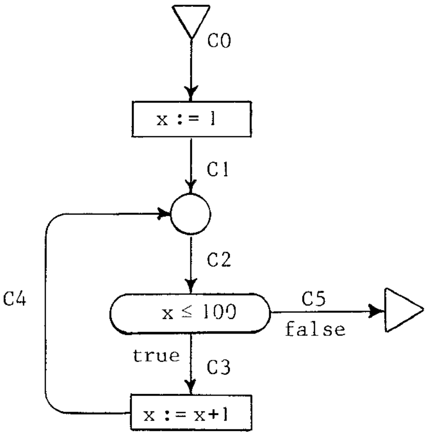
$\Longleftrightarrow\ S_n(q) \leq \underline{\text{Int}}(q, S_n)$

$\Longleftrightarrow\ S_n(q) \leq \underline{\text{A-int}}(q, S_n)$ by 9.1.3.4

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

9.2 Example : Bounds of Integer Variables

In a PASCAL program operating on arrays, the compiler should ensure that arrays are subscripted only by indices within bounds. For that purpose one can use the lattice I_I of section 7. Let us take an obvious example :



Let us note $[a, b]$ where $a \leq b$ the predicate $a \leq x \leq b$. The system of equations corresponding to the example is :

$$
\begin{array}{ll}
(0) & C0 = [\ ,\ ] \\
(1) & C1 = [1, 1] \\
(2) & C2 = C1 \cup C4 \\
(3) & C3 = C2 \cap [-\infty, 100] \\
(4) & C4 = C3 + [1, 1] \\
(5) & C5 = C2 \cap [101, +\infty]
\end{array}
$$

Assignment statements are treated using an interval arithmetic (e.g. $[i, j] + [k, \ell] = [i+k, j+\ell]$ naturally extended to include the case of the empty interval). Similarly tests are treated using an "interval logic". Since there exist infinite Kleene's sequences (e.g. $[\ ,\ ] < [0, 0] < [0, 1] < \ldots < [0, +\infty]$ for the program x := 0 ; <u>while</u> <u>true</u> <u>do</u> 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 ∇ of intervals by :

- $[\ ,\ ]$ is the null element of ∇
- $[i, j]\ \nabla\ [k, \ell] = [\underline{\text{if}}\ k < i\ \underline{\text{then}}\ -\infty\ \underline{\text{else}}\ i\ \underline{\text{fi}},\ \underline{\text{if}}\ \ell > j\ \underline{\text{then}}\ +\infty\ \underline{\text{else}}\ j\ \underline{\text{fi}}]$

∇ satisfies the requirements of 9.1.3 . According to 9.1.3.4 the system of equations is modified by :

(2) $C2 = C2\ \nabla\ (C1 \cup C4)$

The corresponding approximation sequence is :

$$
\begin{array}{lll}
 & Ci & = [\ ,\ ] \text{ for } i \in [0, 5] \\
* & C1 & = [1, 1] \\
 & C2 & = C2\ \nabla\ (C1 \cup C4) \\
 & & = [\ ,\ ]\ \nabla\ ([1, 1] \cup [\ ,\ ]) \\
 & & = [\ ,\ ]\ \nabla\ [1, 1] \\
 & & = [1, 1] \\
 & C3 & = C2 \cap [-\infty, 100] \\
 & & = [1, 1] \cap [-\infty, 100] \\
 & & = [1, 1] \\
 & C4 & = C3 + [1, 1] \\
 & & = [1, 1] + [1, 1] \\
 & & = [2, 2] \\
 & C2 & = C2\ \nabla\ (C1 \cup C4) \\
 & & = [1, 1]\ \nabla\ ([1, 1] \cup [2, 2]) \\
 & & = [1, 1]\ \nabla\ [1, 2] \\
* & C2 & = [1, +\infty] \\
 & C3 & = C2 \cap [-\infty, 100] \\
 & & = [1, +\infty] \cap [-\infty, 100] \\
* & C3 & = [1, 100] \\
 & C4 & = C3 + [1, 1] \\
 & & = [1, 100] + [1, 1] \\
* & C4 & = [2, 101] \\
 & \text{Note} & : C1 \cup C4 = [1, 101] \leq C2 = [1, +\infty] \\
 & & \quad \text{stop on that path.} \\
 & C5 & = C2 \cap [101, +\infty] \\
 & & = [1, +\infty] \cap [101, +\infty] \\
* & C5 & = [101, +\infty] \\
 & \text{exit, stop.} &
\end{array}
$$

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.

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

The ascending approximation sequence leads to an upper bound $S_m = \underline{\widetilde{\text{A-int}}}^m(\tilde{\top})$ of the least fixpoint $\underline{\overline{\text{Cv}}}$ of $\underline{\widetilde{\text{Int}}}$: $\underline{\overline{\text{Cv}}} \tilde{\leq} S_m$. Moreover $\underline{\widetilde{\text{Int}}}(S_m) \tilde{\leq} S_m$. Since $\underline{\widetilde{\text{Int}}}$ is order preserving, this implies that :

$$S_m \tilde{\geq} \underline{\widetilde{\text{Int}}}(S_m) \tilde{\geq} \ldots \tilde{\geq} \underline{\widetilde{\text{Int}}}^n(S_m) \tilde{\geq} \ldots \tilde{\geq} \underline{\overline{\text{Cv}}}.$$

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

9.3.1 *Decreasing Approximation Sequence*

At step n in the descending sequence, we have :

$$\underline{\widetilde{\text{Int}}}^{n-1}(S_m) \tilde{>} \underline{\widetilde{\text{Int}}}^n(S_m) \tilde{>} \underline{\overline{\text{Cv}}}$$

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

$$\underline{\widetilde{\text{Int}}}^{n+1}(S_m) \tilde{>} D \tilde{\geq} \underline{\widetilde{\text{Int}}}(D)$$

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

9.3.2 *Truncated Decreasing Sequence*

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

$$\exists n \geq 0 \mid \underline{\widetilde{\text{Int}}}(S_m) \tilde{>} D \tilde{\geq} \underline{\widetilde{\text{Int}}}^{n+1}(S_m)$$

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

Let $\widetilde{\text{D-int}} : \widetilde{\text{A-Cont}} \to \widetilde{\text{A-Cont}}$ be such that :

$$\begin{array}{ll} 9.3.2.1 & \{\forall C \in \widetilde{\text{A-Cont}}\} \\ & \quad \{C \mathrel{\tilde{>}} \widetilde{\text{Int}}(C)\} \Longrightarrow \{C \mathrel{\tilde{\geq}} \widetilde{\text{D-int}}(C) \mathrel{\tilde{\geq}} \widetilde{\text{Int}}(C)\} \end{array}$$

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

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

$$\begin{array}{ll} 9.3.2.3 & S'_0 = S_m \\ & S'_{n+1} = \text{if } (S'_n \neq \widetilde{\text{Int}}(S'_n)) \text{ and } (S'_n \neq \widetilde{\text{D-int}}(S'_n)) \\ & \qquad\qquad\qquad \text{then } \widetilde{\text{D-int}}(S'_n) \\ & \qquad\quad \text{else} \\ & \qquad\qquad S'_n \\ & \qquad\quad \text{fi} \end{array}$$

Let us now prove that the truncated decreasing sequence is a finite strictly decreasing chain which terms are greater than $\overline{\text{Cv}}$ the least fixpoint of $\widetilde{\text{Int}}$.

Let p be the least natural number (eventually infinite) such that $S'_p = S'_{p+1}$. Trivially from 9.1.1 :

$$S'_0 = S_m \mathrel{\tilde{\geq}} \widetilde{\text{Int}}(S'_0) \mathrel{\tilde{\geq}} \overline{\text{Cv}}$$

If p > 0 then $S'_0 \neq \widetilde{\text{Int}}(S'_0)$, therefore $S'_0 \mathrel{\tilde{>}} \widetilde{\text{Int}}(S'_0)$. Then applying 9.3.2.1 we have :

$$S'_0 \geq \widetilde{\text{D-int}}(S'_0) = S'_1 \mathrel{\tilde{\geq}} \widetilde{\text{Int}}(S'_0) \mathrel{\tilde{\geq}} \overline{\text{Cv}}$$

But 9.3.2.3 implies $S'_0 \neq \widetilde{\text{D-int}}(S'_0)$, hence :

$$S'_0 > S'_1 \mathrel{\tilde{\geq}} \widetilde{\text{Int}}(S'_0) \mathrel{\tilde{\geq}} \overline{\text{Cv}}$$

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

$$S'_{k-1} \mathrel{\tilde{>}} S'_k \mathrel{\tilde{\geq}} \widetilde{\text{Int}}(S'_{k-1}) \mathrel{\tilde{\geq}} \overline{\text{Cv}}$$

Since $\widetilde{\text{Int}}$ is order preserving we have :

$$\widetilde{\text{Int}}(S'_{k-1}) \mathrel{\tilde{\geq}} \widetilde{\text{Int}}(S'_k) \geq \widetilde{\text{Int}}^2(S'_{k-1}) \mathrel{\tilde{\geq}} \widetilde{\text{Int}}(\overline{\text{Cv}}) = \overline{\text{Cv}}$$

By transitivity $S'_k \mathrel{\tilde{\geq}} \widetilde{\text{Int}}(S'_k)$ and since 9.3.2.3 implies $S'_k \neq \widetilde{\text{Int}}(S'_k)$ we have from 9.3.2.1 :

$$S'_k \mathrel{\tilde{\geq}} \widetilde{\text{D-int}}(S'_k) = S'_{k+1} \mathrel{\tilde{\geq}} \widetilde{\text{Int}}(S'_k)$$

Since 9.3.2.3 implies $S'_k \neq \widetilde{\text{D-int}}(S'_k)$ we have :

$$S'_k \mathrel{\tilde{>}} S'_{k+1} \mathrel{\tilde{\geq}} \widetilde{\text{Int}}(S'_k) \mathrel{\tilde{\geq}} \overline{\text{Cv}}$$

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 Descending Sequence

When A-Cont satisfies the descending chain condition, one can choose $\widetilde{\text{D-int}}$ to be $\widetilde{\text{Int}}$, in which case the final result $S'_p = \widetilde{\text{Int}}^p(S_m)$ is a fixpoint greater or equal to the least fixpoint $\overline{\text{Cv}}$ of $\widetilde{\text{Int}}$.

The limit of the descending sequence $S'_0 = \tilde{\top}$, ..., $S'_p = \widetilde{\text{D-int}}^p(\tilde{\top})$, ... is an upper bound of the greatest fixpoint of $\widetilde{\text{Int}}$.

9.3.4 Narrowing in Truncated Decreasing Sequences

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

9.3.4.1 Δ : A-Cont × A-Cont → A-Cont

9.3.4.2 $\forall (C, C') \in \text{A-Cont}^2$,
$\{C \geq C'\} \Longrightarrow \{C \geq C \;\Delta\; C' \geq C'\}$

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

The approximated interpretation D-int : $\text{Arcs}^0 \times \widetilde{\text{A-Cont}} \to$ A-Cont is defined by :

$$\begin{array}{ll} 9.3.4.4 & \text{D-int} = \lambda(q, \text{Cv}) \,.\, \text{if } q \in \text{W-arcs then} \\ & \qquad\qquad\qquad\qquad \text{Cv}(q) \;\Delta\; \text{Int}(q, \text{Cv}) \\ & \qquad\qquad\qquad\quad \text{else} \\ & \qquad\qquad\qquad\qquad \text{Int}(q, \text{Cv}) \\ & \qquad\qquad\qquad\quad \text{fi} \\ & \text{and} \\ & \widetilde{\text{D-int}} = \lambda\text{Cv} \,.\, (\lambda q \,.\, \text{D-int}(q, \text{Cv})) \end{array}$$

This definition of $\widetilde{\text{D-int}}$ trivially satisfies the requirement 9.3.2.1 since $\forall \text{Cv} \in \widetilde{\text{A-Cont}}$ with property $\text{Cv} \mathrel{\tilde{>}} \widetilde{\text{Int}}(\text{Cv})$ implies $\text{Cv}(q) \geq \text{Int}(q, \text{Cv})$, $\forall q \in$ Arcs. If $q \in$ W-arcs then 9.3.4.2 implies that $\text{Cv}(q) \geq \text{Cv}(q) \;\Delta\; \text{Int}(q, \text{Cv}) = \text{D-int}(q, \text{Cv}) \geq \text{Int}(q, \text{Cv})$. Otherwise, if $q \notin$ W-arcs $\text{Cv}(q) \geq \text{Int}(q, \text{Cv}) = \text{D-int}(q, \text{Cv})$. Hence $\text{Cv} \mathrel{\tilde{\geq}} \widetilde{\text{D-int}}(\text{Cv}) \mathrel{\tilde{\geq}} \widetilde{\text{Int}}(\text{Cv})$.

The proof of termination (requirement 9.3.2.2) is very similar to the one outlined for $\widetilde{\text{A-int}}$ in section 9.1.3.

9.4 Example : Bounds of Integer Variables

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

$$\begin{array}{lll} (1) & C1 & = [1, 1] \\ (2) & C2 & = C1 \cup C4 \\ (3) & C3 & = C2 \cap [-\infty, 100] \\ (4) & C4 & = C3 + [1, 1] \\ (5) & C5 & = C2 \cap [101, +\infty] \end{array}$$

The ascending approximation sequence led to the approximate solution :

$$\begin{array}{lll} \star & C1 & = [1, 1] \\ & C2 & = [1, +\infty] \\ & C3 & = [1, 100] \\ \star & C4 & = [2, 101] \\ & C5 & = [101, +\infty] \end{array}$$

Let us define the narrowing Δ of intervals by :

- [,] is the null element of Δ.
- $[i, j] \;\Delta\; [k, \ell] =$
  $[\text{if } i = -\infty \text{ then } k \text{ else } \min(i, k) \text{ fi},$
  $\;\text{if } j = +\infty \text{ then } \ell \text{ else } \max(j, \ell) \text{ fi}]$

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) $C2 = C2 \Delta (C1 \cup C4)$

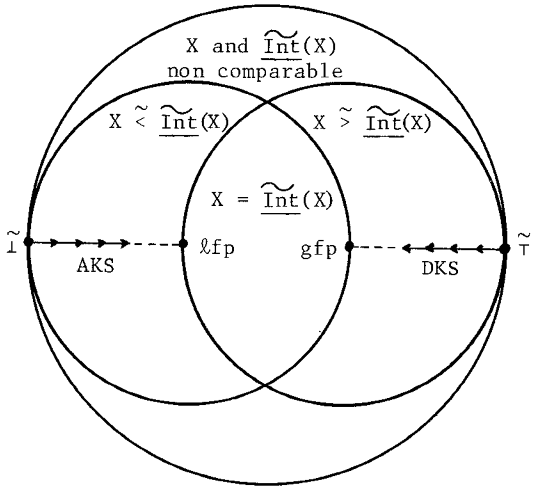
The descending approximation sequence is :

```
    C2 = C2 Δ (C1 ∪ C4)
       = [1, + ∞] Δ ([1, 1] ∪ [2, 101])
       = [1, + ∞] Δ [1, 101]
*   C2 = [1, 101]
    C3 = C2 ∩ [- ∞, 100]
*   C3 = [1, 101] ∩ [- ∞, 100] = [1, 100]
       stop on that path.
    C5 = C2 ∩ [101, + ∞]
*   C5 = [1, 101] ∩ [101, + ∞] = [101, 101]
       exit.
```

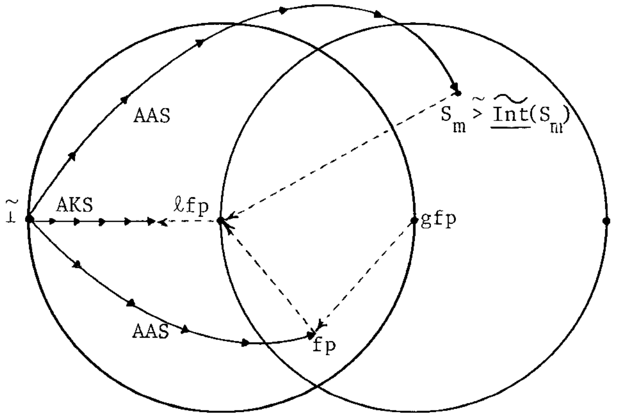
On that example the approximate solution has been improved so that the least fixpoint is reached but this is not the case in general.

9.5 Dual Approximation Methods

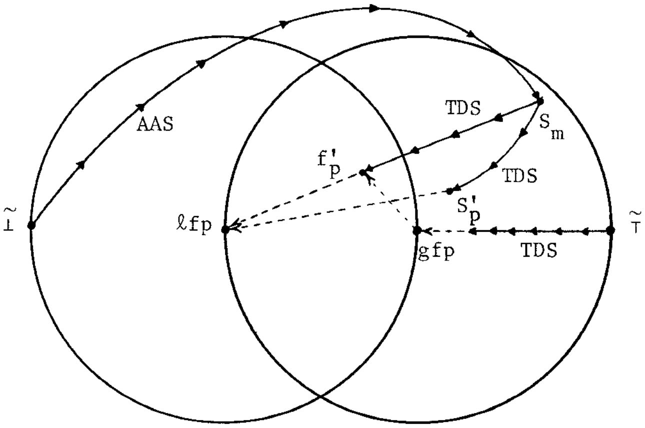
The lattice $\widetilde{A\text{-}Cont}$ may be partitionned as follows :


X and $\underline{\widetilde{Int}}(X)$
non comparable
$X \tilde{<} \underline{\widetilde{Int}}(X)$
$X \tilde{>} \underline{\widetilde{Int}}(X)$
$X = \underline{\widetilde{Int}}(X)$
$\tilde{\bot}$
AKS
ℓfp
gfp
DKS
$\tilde{T}$

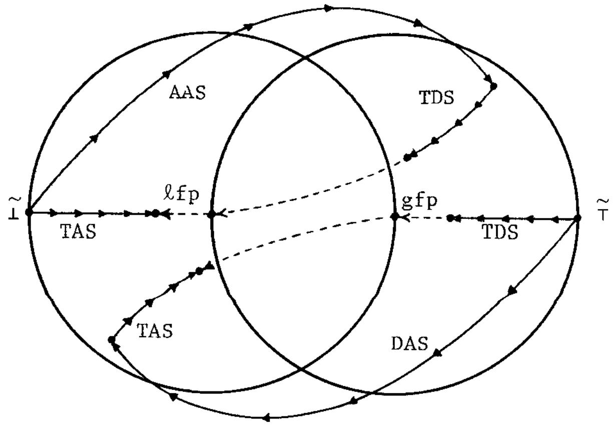
ℓfp and gfp are the least and greatest fixpoints of $\widetilde{Int}$. The ascending (AKS) and descending (DKS) Kleene's sequences converge toward ℓfp and gfp respectively. These limits are reached when $\underline{Int}$ is continuous. When AKS is infinite we have proposed to use an ascending approximation sequence (AAS) to approximate ℓfp. Its limit may be some fixpoint fp, or some S_m such that $S_m \tilde{>} \underline{\widetilde{Int}}(S_m)$ and $S_m \tilde{>}$ ℓfp.



When $X \tilde{\geq} Y$ we have noted X •----->• 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 $\{X \mid X \tilde{\geq} \underline{\widetilde{Int}}(X)\}$, so that their limit S'_p is greater than ℓfp :



It is clear that the ascending approximation sequence AAS when starting from $\tilde{\bot}$ leads to an *upper* bound of the least fixpoint ℓfp of $\widetilde{Int}$, and the truncated descending sequence TDS when starting from $\tilde{T}$ leads to an *upper* bound of the greatest fixpoint gfp.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 $\underline{\widetilde{Int}}$:



Any of the AAS, TDS, DAS, TAS methods may yields a fixpoint fp which is not the fixpoint ℓfp or gfp of interest. None of these methods can improve fp to reach ℓfp or gfp, therefore a "fixpoint improvement method" is necessary. It is our feeling that such a method could be designed only when considering that A-Cont possesses a richer structure (i.e. for particular applications).

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

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

10. *Conclusion*

It is our feeling that most program analysis techniques may be understood as abstract interpretations of programs. Let us point out global data flow analysis in optimizing compilers (Kildall[73], Morel and Renvoise[76] , Schwartz[75], Ullman[75], Wegbreit[75], ...), type verification (Naur[65], ...), type discovery (Cousot[76'], Sintzoff[72], Tenenbaum[74], ...), program testing (Henderson [75], ...) symbolic evaluation of programs (Hewitt et al.[73], Karr[76], ...), program performance analysis (Wegbreit[76], ...), formalization of program semantics (Hoare and Lauer[74], Ligler[75], Manna and Shamir[75], ...), verification of program correctness (Floyd[67], Park[69], Sintzoff[75], ...), discovery of inductive invariants (Katz and Manna[76], ...), proofs of program termination (Sites[74], ...), program transformation (Sintzoff [76], ...), ...

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

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

Acknowledgements

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

11. *References*

Birkhoff[6]. Lattice theory. Amer. Math. Soc. Col. Pub., XXV, Rev. ed.

Cousot[76]. Static determination of dynamic properties of programs. Programming Symp. Paris. Springer-Verlag Lecture Notes in Comp. Sc. to appear (April).

Cousot[76']. Static determination of dynamic properties of generalized type unions. Submitted for publication. (Sept.)

Floyd[67]. Assigning meanings to programs. Proc. Symp. in Appl. Math. Vol. 19. Mathematical Aspects of Computer Science, (J. Schwartz, ed.) AMS, Providence, R.I., 19-32.

Henderson[75]. Finite state modelling in program development. Proc. Int. Conf. on Reliable Soft., Los Angeles, 221-227.

Hewitt et al.[73]. Actor induction and meta-evaluation. Conf. Rec. of the First ACM Symp. on Principles of Programming Languages, Boston, 153-168, (Oct.).

Hoare[67]. An axiomatic basis for computer programming. Comm. ACM 12, 10 (Oct.), 576-580.

Hoare and Lauer[74]. Consistent and Complementary formal theories of the semantics of programming languages. Acta Inf. 3, 135-153.

Kam and Ullman[75]. Monotone data flow analysis frameworks. TR.169, C.S. Lab., Princeton Univ.

Karr[76]. Affine relationships among variables of a program. Acta Inf. 6, 133-151.

Katz and Manna[76]. Logical analysis of programs. Comm. ACM 19, 4(April), 188-206.

Kildall[73].A unified approach to global program optimization. Conf. Rec. of the First ACM Symp. on Principles of Programming Languages, Boston, 194-206, (Oct.).

Kleene[52]. Introduction to metamathematics. Van Nostrand, New York.

Ligler[75]. Surface properties of programming language constructs. Int. Symp. on Proving and Improving Programs,(G. Huet and G. Kahn, eds.), IRIA, France.

Mac Neille[37]. Partially ordered sets. Trans. Amer. Math. Soc., 42, 416-460.

Manna and Shamir[75]. A new approach to recursive programs. Tech. Rep. CS-75-539, Comp. Sc. Dep., Stanford U.

Morel and Renvoise[76]. Une méthode globale d'élimination des redondances partielles. Programming Symp. Paris. Springer-Verlag Lecture Notes in Comp. Sc. to appear.(April).

Naur[65]. Checking of operand types in ALGOL compilers, BIT 5, 151-163.

Park[69]. Fixpoint induction and proofs of program properties. Machine Intelligence 5, (B. Meltzer and D. Michie, eds.), Edinburgh U. Press, 59-78.

Schwartz[75]. Automatic data structure choice in a language of very high level. Comm. ACM 18, 12 (Dec.), 722-728.

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

Scott and Strachey[71]. Towards a mathematical semantics for computer languages. Tech. Mon. PRG-6, Oxford U. Comp. Lab.

Sintzoff[72]. Calculating properties of programs by valuations on specific models. Proc. ACM conf. on Proving Assertions About Programs. SIGPLAN Notices 7, 1, 203-207.

Sintzoff[75]. Vérifications d'assertions pour les fonctions utilisables comme valeurs affectant des variables extérieures. Int. Symp. on Proving and Improving Programs, (G. Huet and G. Kahn, eds.). IRIA. France.

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

Sites[74]. Proving that computer programs terminate cleanly. Ph.D. Th., Comp. Sc. Dep., Stanford U., (May).

Tarjan[75]. Solving path problems on directed graphs. Tech. Rep. CS-75-528, Comp. Sc. Dep., Stanford U.

Tarjan[76]. Iterative algorithms for global flow analysis. Tech. Rep. CS-76-547, Comp. Sc. Dep., Stanford U.

Tarski[55]. A lattice theoretical fixpoint theorem and its applications. Pacific journal of Math. 5, 285- 309.

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

Ullman[75]. Data flow analysis. Tech. Rep. 179, Dep. of Elec. Eng., Comp. Sc. Lab., Princeton U., (March).

Wegbreit[75]. Property extraction in well-founded property sets. IEEE trans. on Soft. Eng., Vol. SE-1, N∘. 3, (Sept.)

Wegbreit[76]. Verifying program performance, J.ACM, 23, 4, (Oct.), 691-699.

12. *Appendix*

We note $<L, \cup, \leq, \top, \bot>$ a complete $\cup$-semilattice L, with partial ordering $\leq$, supremum $\top$ and infimum $\bot$. These definitions are given in Birkhoff[61].

Note : L is a complete lattice.
(proof in Birkhoff[61], p. 49).

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

$$\{\forall (x, y) \in L^2,\ \{x \leq y\} \Longrightarrow \{f(x) \leq f(y)\}\}$$
$$\Longleftrightarrow \{\forall (x, y) \in L^2,\ \{f(x \cup y)\} \geq f(x) \cup f(y)\}\}$$

(H1) : Let F be an order-preserving function from the complete semi-lattice $<L, \cup, \leq, \top, \bot>$ in itself.

$(\overline{\text{H1}})$' : Let $\overline{F}$ be an order-preserving function from the complete semi-lattice $<\overline{L}, \overline{\cup}, \overline{\leq}, \overline{\top}, \overline{\bot}>$ in itself.

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

$$g = \cup\{x \mid (x \in L) \wedge (x \leq F(x))\}$$
$$\ell = \cap\{x \mid (x \in L) \wedge (F(x) \leq x)\}$$

(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 $\overline{g}$ and $\overline{\ell}$ the greatest and least fixpoints of $\overline{F}$.

(H2) : Let α and β be such that :

(H2.1) $\alpha : L \to \overline{L}$
(H2.2) $\gamma : \overline{L} \to L$
(H2.3) α is order preserving
(H2.4) γ is order preserving
(H2.5) $\forall \overline{x} \in \overline{L},\ \overline{x} = \alpha(\gamma(\overline{x}))$
(H2.6) $\forall x \in L,\ x \leq \gamma(\alpha(x))$

(H3.1) : (H1), $(\overline{\text{H1}})$, (H2) and $\{\forall x \in L,\ \overline{F}(\alpha(x)) \mathrel{\overline{\geq}} \alpha(F(x))\}$
(H3.2) : (H1), $(\overline{\text{H1}})$, (H2) and $\{\forall \overline{x} \in \overline{L},\ \gamma(\overline{F}(\overline{x})) \geq F(\gamma(\overline{x}))\}$

(L2) : {H3.1} <==> {H3.2}

Proof :
$\forall \overline{x} \in \overline{L}$,
$\overline{F}(\alpha(\gamma(\overline{x}))) \mathrel{\overline{\geq}} \alpha(F(\gamma(\overline{x})))$ by $x = \gamma(\overline{x})$ in H3.1
$\overline{F}(\overline{x}) \mathrel{\overline{\geq}} \alpha(F(\gamma(\overline{x})))$ from H2.5
$\gamma(\overline{F}(\overline{x})) \geq \gamma(\alpha(F(\gamma(\overline{x}))))$ from H2.4
$\gamma(\overline{F}(\overline{x})) \geq F(\gamma(\overline{x}))$ H2.6 and transitivity.

$\forall x \in L$
$\gamma(\overline{F}(\alpha(x))) \geq F(\gamma(\alpha(x)))$ $\overline{x} = \alpha(x)$ in H3.2
$\gamma(\alpha(x)) \geq x$ H2.6
$F(\gamma(\alpha(x))) \geq F(x)$ F order preserving in (H1).
$\gamma(\overline{F}(\alpha(x))) \geq F(x)$ transitivity
$\alpha(\gamma(\overline{F}(\alpha(x)))) \mathrel{\overline{\geq}} \alpha(F(x))$ H2.3
$\overline{F}(\alpha(x)) \mathrel{\overline{\geq}} \alpha(F(x))$ H2.5

Q.E.D.

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

(H3) : (H3.1) <u>or</u> (H3.2)

(L3) : Let $F : L \to L$ be an order-preserving function from the semilattice $<L, \cup, \leq, \top, \bot>$ in itself, ℓ and g respectively the least and greatest fixpoints of F, then :

$$\forall x \in L,\ \{g \cup F(x) \geq x\} \Longleftrightarrow \{g \geq x\}$$

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

$$\forall x \in L,\ \{\ell \cap F(x) \leq x\} \Longleftrightarrow \{\ell \leq x\}$$

(T1) : H1, $\overline{\text{H1}}$, H2, H3 imply that the greatest fixpoints g and $\bar{g}$ of F and $\bar{F}$ are related by :

$$\{\alpha(g) \mathrel{\bar{\leq}} \bar{g}\} \ \underline{\text{and}}\ \{g \leq \gamma(\bar{g})\}$$

Proof :

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

$\bar{g} \mathbin{\bar{\cup}} \alpha(g) \mathrel{\bar{\geq}} \alpha(g)$	trivially
$\bar{g} \mathbin{\bar{\cup}} \alpha(F(g)) \mathrel{\bar{\geq}} \alpha(g)$	since $g = F(g)$
$\bar{g} \mathbin{\bar{\cup}} \bar{F}(\alpha(g)) \mathrel{\bar{\geq}} \alpha(g)$	H3.1, $\bar{\cup}$ isotone, $\bar{\geq}$ transitive
$\bar{g} \mathrel{\bar{\geq}} \alpha(g)$	L3
$\gamma(\bar{g}) \geq \gamma(\alpha(g))$	H2.4
$\gamma(\bar{g}) \geq g$	H2.6, $\geq$ transitive.

Q.E.D.

Replacing $<g, \bar{g}, \bar{\cup}, \bar{\geq}, \geq, F, \bar{F}, \alpha, \gamma$, H3.1, H2.4, H2.6> respectively by $<\bar{\ell}, \ell, \cap, \leq, \bar{\leq}, \bar{F}, F, \gamma, \alpha$, H3.2, H2.3, H2.5> in the above proof, we get the "dual" theorem :

(T2) : H1, $\overline{\text{H1}}$, H2, H3 imply that the least fixpoints ℓ and $\bar{\ell}$ of F and $\bar{F}$ are related by :

$$\{\gamma(\bar{\ell}) \geq \ell\} \ \underline{\text{and}}\ \{\bar{\ell} \mathrel{\bar{\geq}} \alpha(\ell)\}$$

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

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

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

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

(L4) : H4($\overline{\text{H4}}$) implies that F ($\bar{F}$) has a least fixpoint $\ell(\bar{\ell})$ which is the limit $\overset{+\infty}{\underset{i=0}{\cup}} F^i(\bot)$ of the Kleene's sequence $\bot \leq F(\bot) \leq \ldots \leq F^n(\bot) \leq \ldots$

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