Denotational definition of properties of programs computations

Veronique Donzeau-Gouge

IRIA
laboria

Institut de Recherche
d'Informatique
et d'Automatique

Domaine de Voluceau
Rocquencourt
B. P. 105 78150 - Le Chesnay
France
Tél.: 954 90 20

laboratoire de recherche
en informatique
et automatique

DENOTATIONAL DEFINITION OF PROPERTIES OF PROGRAMS COMPUTATIONS

Véronique DONZEAU-GOUGE

Rapport de Recherche N° 349

Avril 1979

DENOTATIONAL DEFINITION OF PROPERTIES OF PROGRAMS COMPUTATIONS

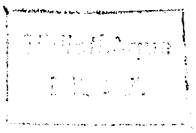
Véronique Donzeau-Gouge
IRIA

Résumé :

Ce rapport présente une définition dénotationnelle de propriétés associées au comportement des programmes. Cette définition testée par machine et indépendante de tout algorithme utilisé pour le calcul de ces propriétés permet de prouver une relation de cohérence liant ces propriétés aux exécutions des programmes.

Abstract :

We present a denotational machine-checked definition of properties associated to programs computations. This formal definition, independent of any algorithm, is used to prove a consistency relation between properties and computations.



INTRODUCTION

Syntactic transformations of a program may be done if properties have been defined and associated with the elements of the program.

These properties characterize the behavior of the program's computations.

For instance let us assume that the expression x+2 of a program prog is replaced by the syntactic constant 5. The transformation is valid if the initial program prog and the modified one prog' are strongly equivalent. This can be proved if a property associated with the identifier x specifies that its value is constant and equal to 3 for each computation of prog.

Properties of program computations can be defined by non standard interpretations (Cousot [2], Sintzoff [14], Weghreit [18]), in other words, by interpretations defined on domains suitably constructed to model the desired properties.

This notion of non-standard interpretation exists also in arithmetic : "casting out the nines" can be understood as a non-standard interpretation of the multiplication operation. It gives information about the correctness of the multiplication. In physics, calculation of the dimension of a formula gives its unit

The properties defined for a program by non-standard interpretations must be satisfied by any computation described by the standard interpretation of this program. To verify the coherence, we must define and prove relations between these interpretations. For instance the relation between casting out nines and multiplication uses properties of the modulo operation.

Such comparisons of interpretations may be done more easily if mathematical descriptions of the interpretations are used.

Using the work of C. Strachey and D. Scott, we can define an interpretation as a function from input to output domains, where a domain is a partially ordered set in which the partial ordering models the notion of approximation in sequences of computations. An undefined

element is included so that only total functions need to be considered ; completeness properties ensure the existence of limits for sequences of computed objects.

This denotational semantics gives an abstract definition of each property and this definition is independent of any algorithms used to compute this property. Using this abstraction we prove a congruence between non-standard and standard interpretations.

Properties of the behavior of program computations have often been studied in the literature as "flow analysis problems"; this work is applied to classical flow analysis.

We consider data flow analysis used to compute properties of the values of expressions :

- Constant propagation : an occurrence of an expression is a constant if, for each computation of the program, the value of this occurrence is a known constant. The problem is to find such occurrences and the associated constant value.

- Determination of common sub-expressions : the problem is to associate with each occurrence of an expression the set of expressions which have same value in any computations as the one considered. This problem includes the determination of available expressions (an expression is available if its value has been previously computed and since the last evaluation of the expression, no identifier appearing in the expression has had its value changed).

- Scalar propagation : same problem than the previous one restricted to the set of identifiers which have, in any computations, the same value as the expression at the occurrence considered.

- Determination of invariant expressions : an expression is invariant in a computation if its value remains constant during this computation : the goal is to determine the expressions which are invariant in any computations.

We use to illustrate this study a simple Algol like language. A larger language could be managed by a similar method but the proof would be longer and more tedious.

The definition of the different denotational interpretations of this simple language are checked using the system SIS [10] built by P. Mosses.

The concrete and associated abstract syntax of the simple language are written in GRAM [10].

The language DSL [10] is used to write the semantic functions.

We expect that the reader as a good understanding of denotational semantics [16] and of its basic concepts of curryfication and continuation.

A knowledge of the S I S system and of the syntaxe of the G R A M and D S L languages is usefull but not necessary : these languages are very closed from respectivelly B N F and usual notations used in denotational semantics.

The plan of this study is the following :

1) Introduction to some basic concepts used in denotational semantics.

2) Specification of the concrete and abstract syntax of the simple language.

3) Presentation of the notion of occurrences : it is a formalization of the notion of point of program.

4) Formal definition including occurrence of the standard semantics of the simple language.

5) Definition of the domains and equations used in a denotational definition of properties and presentation of a formal definition of properties associated with expressions.

6) Proof of the coherence of the non-standard interpretation and the standard-one.

7) Applications to some classical data flow analysis problems.

1) DENOTATIONAL SEMANTICS

Based on the works of C. Strachey and D. Scott, this approach to semantics associates with each program a total function defined from input data to answers, where :

a domain is a partially ordered set $<D,\sqsubseteq>$ with an element $\perp$ and a binary operation $\sqcup$ such that [13] :

1- $\perp \sqsubseteq d \qquad d \in D$

2- $d_1,d_2,d_3 \in D \quad d_1,d_2 \sqsubseteq d_3$ imply $d_1 \sqcup d_2$ is defined

3- $d_1 \sqsubseteq d_1 \sqcup d_2$ and $d_2 \sqsubseteq d_1 \sqcup d_2$

4- $d_1,d_2 \sqsubseteq d_3$ imply $d_1 \sqcup d_2 \sqsubseteq d_3$

The existence of a limit for a list of computed elements is ensured by the notion of completness :

a domain D is complete if

- each subset S of D is directed ie

  $S \neq \emptyset$ and $\forall s_1 s_2 \in S \quad \exists s_3 \in S \quad s_1 \sqsubseteq s_3$ and $s_2 \sqsubseteq s_3$

- there exist $\sqcup S \in D$ such that :

  1- $\forall s \in S \quad s \sqsubseteq \sqcup S$

  2- $s \sqsubseteq d$ imply $\sqcup S \sqsubseteq d \quad \forall s \in S$

Operations can be defined on domains. It is proved in [12] that if D and D' are domains, so are :

- the product $<D,D'> = \{<d,d'> | d \in D, d' \in D'\}$
- the sum $D/D' = \{<1,d> | d \in D\} \cup \{<2,d'> | d' \in D'\} \cup \{\perp\}$
- $D \rightarrow D'$ the domain of continuous functions from D to D'
- D^* the domain of finite sequences of elements of D :

  $D^* = <>/<D,D^*>$

We say that s is an upper bound of S iff :

$\forall s' \quad s' \in S \quad s' \sqsubseteq s$

Two elements d_1, d_2 which belong to D are compatible if $\{d_1, d_2\}$ has an upper bound.

The denotational semantics of a programming language associates which each syntactic construct a function which models its meaning in terms of the meaning of its components.

In fact a semantic function is associated with each syntactic domain which is defined by the abstract syntax of the programming language.

2) SYNTAX OF A SIMPLE LANGUAGE

To illustrate this study we define a simple Algol like language.

The semantics is defined on the abstract syntax of the language : it maps trees to denotations. Thus it is necessary to specify the correspondence between the concrete strings defined by the syntactic rules and the abstract syntax trees.

We give in table 1 a formal definition of the syntax of the simple language together with a corresponding abstract syntax.

Using this definition, every concrete program written in this simple language can be translated into its abstract representation.

The intuitive meaning of each production is clear .

When an expression of the form "valof" stm-train "end" is encountered, the statement part is executed until a form "resultis" exp is reached. The control passes out of the smallest "valof" expression containing this command. The value of exp is taken to be the value of the original expression.

The remaining expressions are more familiar.

The lexical part of the specification of the simple language does not require any particular comment.

Some GRAM notations need to be describe , the reader must refer to [10] for more details :

- a production has a non terminal to the left of "::=" and a list of alternatives separated by "/" to the right ; it is terminated by a ";".
- the iterator item 1 +- item 2 allows one or more occurrences of item 1, separated by occurrences of item 2.

Table 1

```
CRAM "simple"
!*************************************************
 SYNTAX                                          !*
!*************************************************

program ::=  "begin"  stm-train "end"   ;

! [Statemants]
!-------------

stm-train  ::= statement+-";"  ;


statement ::= asgt-stm   :asgt-stm /
              cond-stm   :cond-stm /
              loop-stm   :loop-stm /
              transput-stm :transput-stm   /
              result-stm :result-stm ;

result-stm ::= "resultis" exp ;


asgt-stm  ::= id ":=" exp  ;

cond-stm  ::= "if" exp "then" stm-train "fi" /
              "if" exp "then" stm-train "else" stm-train "fi"  ;

loop-stm  ::= "while" exp "do" stm-train "end" ;

transput-stm ::= "input" id   /
                 "output" exp mode ;

mode      ::= "int" :"int"/
              "bool":"bool" ;


! [Expressions]
!--------------

exp       ::= factor:factor  /
              exp add-op factor ;

factor    ::= primary:primary    /
              factor mult-op primary ;


primary   ::= id : id             /
              constant : constant /
              "(" exp ")" : exp   /
              "(" compare ")" : compare /
             "valof" stm-train "end" ;

compare   ::= exp rel-op exp ;
```

```
! [Constants and Identifiers]
!----------------------------

add-op === "+" / "or" ;
mult-op === "*" / "and" ;
rel-op === "=" / " #"  ;

id         ::= ident  ;


constant   ::= bool   /
               num    ;

bool       ::= "true" : TT /
               "false": FF ;

num        ::= "NUM" n : n ;

ident      ::=  "ID" q : q ;

!**********
 DOMAINS !*
!**********
mult-op ,add-op ,rel-op: Op;


stm-train,statement: Stm ;

exp,factor,primary,compare : Exp;

!*************************************************
 LEXIS                                          !*
!*************************************************


program  ::= word+  : (CONC word+ )  ;

word     ::= layoutchar+ : <>         /
             identifier  : <OUT "ID", identifier > /
             numeral     : <OUT "NUM",numeral    > ;

layoutchar === " "/ CC"C" / CC"L" ;

identifier ::= letter+ : (QUOTE letter+ ) ;

letter  === "A"..."Z"  ;

numeral ::= digit+  : (NUMBER digit+) ;

digit   === "0"..."9" ;

END
```

3) OCCURRENCES

We want to define properties associated with elements of programs thus we must be able to name these elements.

In a standard denotational definition, the semantic function follows the structure of the syntactic domain on which it is defined : the function which defines an interpretation of a program follows the syntactic structure of the program ; but it doesn't allow to name elements of this structure.

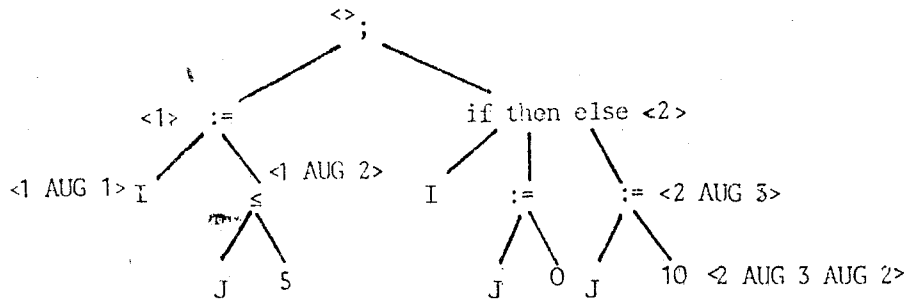
For doing this, we formalize the intuitive notion of an access path.

3.1) Notion of occurrence

A program can be viewed abstractly as a labeled tree. The set of rules that define the abstract syntax of the language describes the set of well-formed trees. Each node is an operator of fixed arity, so that the program :

```
begin
   I := J≤5 ;
  if I then J := 0
       else J := 10
  fi
end
```

is denoted by :



Let N^* be the set of finite sequences of positive integers, <> the empty sequence on N^*, AUG the right concatenation of sequences.

We call occurrences the elements of N^* ; let occ be the meta-variable defined on N^*.

For each tree t, let Occ(t) be the set of occurrences of t and if occ belongs to Occ(t), "t at occ" is the subtree of t at occ. In the example "t at <2 AUG 3>" is the subtree which denotes "J := 10".

Using occurrences we can access and name all subtrees of a tree ie each element of a program.

3.2) Occurrences and denotational semantics

Introducing occurrences in semantic functions allows us to attach a meaning to a specific element of a given program.

Semantic functions are now defined by :

```
     run : Prog  →  Occ  →  Semantic domains
  locate :   Id  →  Occ  →  Semantic domains
evaluate :  Exp  →  Occ  →  Semantic domains
 execute : Stm   →  Occ  →  Semantic domains
```

The computation of the occurrences follows the syntactic structure of the program, thus it follows the recursive structure of the semantic functions and it appears explicitely in the equations.

For instance the equation associated with a sequence of statements becomes :

```
exec [[ stm1; stm2 ]] =
                    LAM  occ.
                      exec [[ stm 1 ]] occ AUG 1 ;
                      exec [[ stm 2 ]] occ AUG 2
```

This can be paraphrased like :

The semantics of a sequence of statements stm1; stm2 is a function of an occurrence occ which applies the semantic function exec to the first statement of the sequence and to the current occurrence occ concatenated

with 1, then applies the function exec to the second element of the sequence and to the current occurrence concatenated with 2.

4) DENOTATIONAL SEMANTICS OF THE SIMPLE LANGUAGE

The formal semantics of the language is presented in table 2 and table 3. It is written in DSL [10] and structured into two parts :

1- an abstract machine in which basic functions are defined (table 3).

2- a function semant parameterized by the abstract machine, which associates to every abstract program a function from an input stream to an output stream (table 2).

4.1) The function semant :

The syntactic and semantic domains are self explanatory. Note that the "valof" expressions can be nested and the matching between "resultis" statements and "valof" expressions is done textually. Hence we make environments equal to expression continuations : Env = Eval-cont.

The domains of continuations are defined as usual. A value can be a location value (in this simple language it can only be an identifier) or a storable value (here a boolean or an integer). The domains of answers, states and inputs are defined in the abstract machine.

The statements and the expressions are understood in terms of semantic evaluations : a function maps the syntactic domains into semantic domains.

Thus :

evaluate is such that :

evaluate := Exp → Occ → Env → Eval-cont → Exec-cont ;

Its application to a syntactic expression, its occurrence and a current environment maps into a transformation from expression continuation to command continuation : both the evaluation of the expression and the side effects are incorporated in this transformation.

The function locate is similarly defined :

locate := Id → Occ → Env → Eval-cont → Exec-cont

The function execute has functionality :

execute := Stm → Occ → Env → Exec-cont → Exec-cont

A program is denoted by :

run := Prog → Exec-cont.

4.1.1) Programs

The list of statements which composes a program are interpreted with the initial environment and finally the null continuation is supplied.

4.1.2) Statements

The interpretation of a list of statements is defined by the auxiliary function "execute-list" applied to the current occurrence, current environment and current continuation.

The execution of a while statement is described with a continuation exec-cont' which is recursively defined. The semantic definition of the conditional is familiar. Input and output statements use in their definition the "read" and "write" functions described in the machine.

The assignment statement first evaluates the location value of its left hand side, then evaluates the right hand side expression in the current environment, updates the left value with the right value and carries on executing the current continuation.

The execution of a "resultis" statement evaluates the expression in the current environment with as continuation the one included in the environment.

4.1.3) Expressions

The "valof" expression modifies the current environment by the current continuation and interprets the statement part. If no "resultis" statement is encountered, the current continuation is replaced by the constant function "wrong" which writes an error message and stops the execution.

4.2) Abstract machine

In this part the low level functions used in the denotational definition are described. They model state handling and input-output transactions. As in [4] input and output are slightly asymetrical. LAMB constants are read and strings of characters are output.

Table 2

```
DSL "SIMPLE semantics"
DOMAINS

! SYNTACTIC VARIABLES AND DOMAINS
! ********************************

prog : Prog = ["begin" Stm "end"] ;
stm : Stm =   [ Stm+ ]
            / [ Id ":=" Exp ]
            / ["if" Exp "then" Stm "fi"]
            / ["if" Exp "then" Stm "else" Stm "fi"]
            / ["while" Exp "do" Stm "end"]
            / ["input" Id ]
            / ["output" Exp Mode]
            / ["resultis" Exp ]
          ;
mode : Mode = "int" / "bool" ;

exp : Exp =   [ Exp Op Exp ]
            / ["valof" Stm "end"]
            / Id
            / [Num]
            / [Bool]
            ;
id    : Id = [Ident]    ;
ident: Ident = Q ;
num   : Num  = N  ;
bool : Bool = T ;
op    : Op   = "+" / "*" / "or" / "and" / "=" / "#" ;


!SEMANTIC VARIABLES AND DOMAINS
!*******************************

answ : Answ ;                                          !Answers
occ  : Occ = N* ;                                      !Occurrences
env : Env = Eval-cont ;                                !Environments
exec-cont : Exec-cont = State -> Input* -> Answ;       !Command continuations
value : Value = R-value / L-value ;
r-value : R-value = N / T ;                            !Storable values
l-value : L-value = Ident ;
eval-cont : Eval-cont = Value  -> Exec-cont;           !Expressions continuations
state : State;                                         !States
input :Input;                                          !Inputs
```

```
!TYPES OF SEMANTIC FUNCTIONS
!**************************

run := Prog ->  Exec-cont;
execute := Stm  -> Occ -> Env -> Exec-cont -> Exec-cont;
execute-list := Stm* -> Occ -> Env -> Exec-cont -> Exec-cont;
evaluate := Exp  -> Occ -> Env -> Eval-cont -> Exec-cont;
locate := Id -> Occ -> Env -> Eval-cont -> Exec-cont;
no-action  := Exec-cont -> Exec-cont;


IN

LAM prog.

!PRIMITIVES
!**********

LAM <
        init-state   : State,
        maxint    : N,
        wrong     : (Q -> Exec-cont),
        update    : ((Ident,R-value) -> Exec-cont -> Exec-cont),
        content   : (Ident -> Eval-cont -> Exec-cont),
        write     : ((R-value,Mode) -> Exec-cont -> Exec-cont),
        read      : (Ident -> Eval-cont -> Exec-cont),
      op-val    : ((R-value,R-value,Op) -> Eval-cont -> Exec-cont)
   >.

!FUNCTIONS DEFINITIONS
!*********************


LET no-action : (Exec-cont ->Exec-cont ) = LAM exec-cont . exec-cont   !Identity on continuations
ALSO init-env : Env =  LAM value. init-exec-cont                !Empty environment
ALSO init-exec-cont : Exec-cont =
                            LAM state. LAM input*. <>                   !Null continuation
LSO init-occ : Occ = <>                                    !Initial occurrences
```

```
DEF
        !PROGRAM
        !*******

        run ["begin" stm "end"] :Exec-cont =
                  execute(stm)(init-occ )init-env;
                  init-exec-cont


        WITH
        !EXPRESSIONS
        !***********

        evaluate (exp)(occ)env;eval-cont :Exec-cont =
                  CASE exp
                  /[ident] ->
                              content(ident);
                              eval-cont
                  /[num] ->
                             LET exec-cont=eval-cont(num)
                             IN exec-cont
                  /[bool]  ->
                             LET exec-cont=eval-cont(bool)
                             IN exec-cont

                  /[exp1 op exp2] ->
                             evaluate(exp1)(occ AUG 1)env; LAM r-value1.
                             evaluate(exp2)(occ AUG 2)env; LAM r-value2.
                             op-val(r-value1,r-value2,op);
                             eval-cont

                  /["valof" stm "end"] ->
                             LET env' = eval-cont
                             IN execute(stm)(occ AUG 1)env';
                             wrong"resultis missing"
                  ESAC
        WITH
        !IDENTIFIERS
        !***********

        locate(id)(occ)env;eval-cont:Exec-cont =

                  LET [ident] = id IN
                   eval-cont(ident)
```

```
WITH
!STATEMENTS
!*********

execute(stm)(occ)env;exec-cont :Exec-cont =
          CASE stm
          /[stm+] -> execute-list(stm+)(occ)env;
                     exec-cont

          /["if" exp "then" stm1 "fi"] ->
                    evaluate(exp)(occ AUG 1)env; LAM t.
                    (t -> execute(stm1)(occ AUG 2)env , no-action);
                    exec-cont

          /["if" exp "then" stm1 "else" stm2 "fi"] ->
                    evaluate(exp)(occ AUG 1)env; LAM t.
                    (t -> execute(stm1)(occ AUG 2)env  , execute(stm2)(occ AUG 3)env);
                    exec-cont


          /["while" exp "do" stm "end"] ->
                    DEF exec-cont'=
                      evaluate(exp)(occ AUG 1)env; LAM t.
                      (t -> execute(stm)(occ AUG 2)env;exec-cont' , exec-cont)
                    IN exec-cont'

           /["input" id ] ->
                    locate(id)(occ AUG 1)env; LAM ident.
                    read(ident); LAM r-value.
                    update(ident,r-value);
                    exec-cont


          /["output" exp mode] ->
                    evaluate(exp)(occ AUG 1)env; LAM r-value.
                    write(r-value,mode);
                    exec-cont

          /[id ":=" exp] ->
                     locate(id)(occ AUG 1)env; LAM ident.
                     evaluate(exp)(occ AUG 2)env; LAM r-value.
                     update(ident,r-value);
                     exec-cont

          /["resultis" exp] ->
                    LET eval-cont = env
                    IN evaluate(exp)(occ AUG 1)(env);
                       eval-cont
          ESAC
          WITH
          !auxiliary functions for statements
          !----------------------------------

          execute-list(stm*)(occ)env;exec-cont :Exec-cont =
          CASE stm*
          /stm1 PRE stm2* ->
                    execute(stm1)(occ AUG 1)env;
                    execute-list(stm2*)(occ AUG 2)env;
                    exec-cont
          /<> ->    exec-cont
          ESAC
          IN
          run (prog) (init-state) : (Input*  -> Answ)

END
```

```
DSL "machine"

DOMAINS

!SEMANTIC VARIABLES AND DOMAINS
!******************************
input     : Input = N / T;                                  !Inputs
answ      : Answ =  Q*;                                     !Answers
mode      : Mode = "int" / "bool";                          !For input/output
r-value   : R-value = N / T ;                               !Storable values
state     : State = Ident -> R-value;                       !States
exec-cont : Exec-cont = State -> (Input* -> Answ);          !Statement continuations
eval-cont : Eval-cont = Value -> Exec-cont;                 !Expressions continuations

!TYPES OF FUNCTIONS
!******************
init-state := State;                                        !Initial state
maxint    := N;                                             !Maximal integer
wrong     := Q -> Exec-cont;
update    := <Ident,R-value> -> Exec-cont -> Exec-cont;
content   := Ident -> Eval-cont -> Exec-cont;
write     := <R-value,Mode> -> Exec-cont -> Exec-ont;
read      := Ident -> Eval-cont -> Exec-cont;
op-val    := <R-value,R-value,Op> -> Eval-cont -> Exec-cont;


IN
!DESCRIPTION OF FUNCTIONS
!************************

LET       maxint :N = 10000

ALSO      wrong(q)(state)(input*) :Answ =
                  <QUOTE<"ERROR : ",q>>
!State handling
!--------------
ALSO      initial-state :State =
                   LAM ident.?

ALSO      update(ident,r-value)(exec-cont)(state) :(Input* -> Answ) =
                  LET state' = state \ ident<- r-value
                  IN exec-cont(state')

LET       content(ident)(eval-cont)(state) :(Input* -> Answ) =
                  LET r-value = state(ident)
                  IN eval-cont(r-value)(state)
!Input-output
!------------
ALSO      write(r-value,mode)(exec-cont)(state)(input*) :Answ =
                  LET q = CASE mode
                          /"int" ->   LET NUMBER q'* = r-value
                                      IN QUOTE q'*
                          /"bool" ->  (r-value -> "true","false")
                          ESAC
                  IN <q> CAT (exec-cont(state)(input*))

ALSO      read(ident)(eval-cont)(state)(input*) :Answ =
                   CASE input*
                   /input1 PRE input2* ->
                           eval-cont(input1)(state)(input2*)
                   /<> -> wrong"end of file"(state)(<>)
                   ESAC
```

```
ALSO    op-val(r-value1,r-value2,op)(eval-cont)(state) :(Input* -> Answ) =
                  CASE op
                  /"+"   ->
                              LET n= r-value1 PLUS r-value2
                              IN n LE maxint -> eval-cont(n), wrong"overflow"
                 /"*"    ->
                              LET n= r-value1 MULT r-value2
                              IN n LE maxint -> eval-cont(n), wrong"overflow"
                  /"or"  ->
                              LET t= r-value1 OR r-value2
                              IN eval-cont(t)
                  /"and" ->
                              LET t= r-value1 AND r-value2
                              IN eval-cont(t)
                  /"="   ->
                              r-value1 EQ r-value2 -> eval-cont(TT),
                                                      eval-cont(FF)
                  /"#"   ->
                              r-value1 NE r-value2 -> eval-cont(TT),
                                                      eval-cont(FF)
                  ESAC

IN <initial-state,maxint,wrong,update,content,write,read,op-val>

END
```

5) DENOTATIONAL DEFINITION OF PROPERTIES ASSOCIATED WITH ELEMENTS OF A PROGRAM

Using the notion of occurrence , we can now define properties associated with elements of a program.

These properties can be defined by non-standard interpretations. The denotational definition of these interpretations gives a mathematical specification of the various algorithms which are used to compute such properties.

5.1) Domains and functions involved in the denotational definition of the properties

The result of a non-standard interpretation associates properties with elements that are indexed by occurrences.

Let Prop the domain of properties

Let Answ the domain of answers

Answ is defined by : Occ → Prop

Properties are associated with elements which belong to a same syntactic domain. Let Elem be this domain, it will be specified for each application.

Let R-value be the domain of basic value of the non-standard interpretation.

The properties are defined from the basic values by the function :

prop-def := R-value → State → Prop

which will be specified in the description of each application.

For example in the problem of constant propagation :

Elem is the syntactic domain Exp

R-value is the domain : N/T

Prop is the domain : R-value/{ var }

As usual a state associates to each element a basic value, thus the domain State is defined by : Elem → R-value.

A continuation function applied to a state and an input stream defines a final answer :

Continuation : State → $Input^*$ → Answ.

Let "continue" be meta-variable defined on the domain of continuations.

A function "update" modifies the current state :

update := <Elem,R-value> → Continuation → Continuation

It is defined by :

```
update (elem,r-value) ; continue : Continuation =
    LAM state.
    LET state' = state\elem ← r-value
     IN continue (state')
```

In an answer the property associated with an occurrence must be compatible with the current property defined from the current computed value, each time the control reaches this occurrence.

This condition can be expressed by the function

attach := <Occ,R-value> → Continuation → Continuation

defined by :

```
attach (occ,r-value) ; continue : Continuation =
LAM state . LAM input*.
    LET prop = prop-def (r-value)
    LET answ = continue (state) (input*)
    LET prop' = answ (occ)
    LET prop '' = compatible (prop,prop')
     IN answ \ occ ← prop''.
```

A current property prop is obtained from the current r-value by the function prop-def. Each new approximation of the final answer associates with the current occurrence occ a property prop'' which is compatible with the current property prop and with the property prop' associated with occ in the current answer.

The function "compatible" applied to prop and prop' gives as result the upper bound of prop and prop' in Prop if prop and prop' are compatible ; if not its results is an element of Prop which cannot be compare with prop neither prop'.

The definition of this function depends on the definition of the domain Prop.

It corresponds to the lattice operation used in classical flow analysis : Kildall [8] , Kam, Ullman [7] , Hecht [6] .

In this kind of non-standard interpretation, the value of a boolean expression may not be a boolean constant. The alternatives of a conditional statement are interpreted in parallel and the final answer is the one which is compatible with the answers resulting from these parallel interpretations.

This operation is defined by the function :

joint := <Continuation,Continuation> → Continuation

described by :

```
joint (continue',continue'') : Continuation =
LAM state . LAM input*.

    LAM occ.
            LET answ' = continue' (state) (input*)
            LET answ'' = continue'' (state) (input*)
             IN
            compatible (answ'(occ),answ''(occ))
```

5.2) Application

We give in table 4 a denotational definition of properties for the simple language used all along this study : the syntactic domains and meta-variables used are the same as the one described in table 2.

In this application, properties are associated with expressions ; hence the semantic domain L-value is equal to the syntactic domain Exp, the domain Prop is introduced and Answ is modified to Occ $\longrightarrow$ Prop.

The function "evaluate" of type ;

Exp $\longrightarrow$ Occ $\longrightarrow$ Env $\longrightarrow$ Eval-cont

can be paraphrased by :

first compute the current value "r- value" of the current expression using the current state,

then define a new approximation of the final answer (function "attach")

then modify the current state (function "update")

and at least apply the current continuation.

The function execute is modified in the definition of the conditional statement which uses the function joint.

The functions locate and run are not changed from the standard interpretation.

This non-standard semantic function is parameterized by the domains of properties, of values of input stream and by the basic functions defined on these domains.

Their specification entirely defines a desired property associated with expressions.

```
DSL "SIMPLE properties"
DOMAINS

!  SYNTACTIC VARIABLES AND DOMAINS
!  *******************************

prog : Prog = ["begin" Stm "end"] ;
stm : Stm =    [ Stm+ ]
             / [ Id ":=" Exp ]
             / ["if" Exp "then" Stm "fi"]
             / ["if" Exp "then" Stm "else" Stm "fi"]
             / ["while" Exp "do" Stm "end"]
             / ["input" Id ]
             / ["output" Exp Mode]
             / ["resultis" Exp ]
           ;
mode : Mode = "int" / "bool" ;

exp : Exp =    [ Exp Op Exp ]
             / ["valof" Stm "end"]
             / Id
             / [Num]
             / [Bool]
             ;
id   : Id = [Ident]   ;
ident: Ident = Q ;
num  : Num = N  ;
bool : Bool = T ;
op   : Op   = "+" / "*" / "or" / "and" / "=" / "#" ;


!SEMANTIC VARIABLES AND DOMAINS
!*******************************

prop : Prop;
answ : Answ = Occ -> Prop ;                          !Answers
occ  : Occ = N* ;                                    !Occurrences
env : Env = Eval-cont ;                              !Environments
exec-cont : Exec-cont = State  -> Input* -> Answ;    !Command continuations
value : Value = R-value / L-value ;
r-value : R-value ;
l-value : L-value = Exp ;
eval-cont : Eval-cont = Value  -> Exec-cont;         !Expressions continuations
state : State= Exp -> R-value ;                       !States
input :Input;                                        !Inputs
```

```
!TYPES OF SEMANTIC FUNCTIONS
!**************************

run := Prog ->  Exec-cont;
execute := Stm  -> Occ -> Env -> Exec-cont -> Exec-cont;
execute-list := Stm* -> Occ -> Env -> Exec-cont -> Exec-cont;
evaluate := Exp  -> Occ -> Env -> Eval-cont -> Exec-cont;
locate := Id -> Occ -> Env -> Eval-cont -> Exec-cont;
no-action  := Exec-cont -> Exec-cont;
joint := <Exec-cont,Exec-cont> -> Exec-cont;
attach := <Occ,R-value> -> Exec-cont -> Exec-cont;
update := <Exp,State> -> Exec-cont -> Exec-cont;
content := L-value ->Eval-cont -> Exec-cont;
wrong   := Exec-cont;


IN

LAM prog.

!PRIMITIVES
!**********

LAM <
       init-prop  :(Prop),
       read       : (Ident -> Eval-cont -> Exec-cont),
       compatible : ((Prop,Prop) -> Prop ),
       prop-def: (R-value -> State -> Prop),
       op-val  : ((R-value,R-value,Op) -> R-value)
   >.

!FUNCTIONS DEFINITIONS
**********************


LET init-answ : Answ = LAM occ. init-prop
ALSO init-state : State = LAM exp. ?
ALSO init-env : Env =  LAM value. init-exec-cont                    !Empty environment
ALSO init-exec-cont : Exec-cont =
                          LAM state. LAM input*. init-answ           !Null continuation
LSO init-occ : Occ = <>                                  !Initial occurrences
LSO  wrong : Exec-cont = init-exec-cont
DEF
       !PROGRAM
       !*******

       run ["begin"  stm "end"] :Exec-cont =
                 execute(stm)(init-occ)init-env;
                 init-exec-cont
```

```
WITH
!EXPRESSIONS
!***********

evaluate (exp)(occ)env;eval-cont :Exec-cont =
          CASE exp
          /[ident] ->
                    content(ident);LAM r-value.
                    attach(occ,r-value);
                    update(exp,r-value);
                    eval-cont(r-value)

          /[num] ->
                    attach(occ,num);
                    update(exp,num);
                    eval-cont(num)
          /[bool]  ->
                    attach(occ,bool);
                    update(exp,bool);
                    eval-cont(bool)

          /[exp1 op exp2] ->
                    evaluate(exp1)(occ AUG 1)env; LAM r-value1.
                    evaluate(exp2)(occ AUG 2)env; LAM r-value2.
                    LET r-value=op-val(r-value1,r-value2,op)
                    IN        attach(occ,r-value);
                              update(exp,r-value);
                              eval-cont(r-value)

          /["valof" stm "end"] ->
                    LET env'=
                       LAM r-value.
                              attach(occ,r-value);
                              update(exp,r-value);
                              eval-cont(r-value)
                    IN        execute(stm)(occ AUG 1)env';
                              wrong
          ESAC
WITH
 !Auxiliary functions for defining properties
 !-------------------------------------------
 attach(occ,r-value);exec-cont: Exec-cont =
          LAM state. LAM input*.
          LET answ=exec-cont(state)(input*)
          ALSO prop = prop-def(r-value)(state)
          IN
                    answ\occ <- compatible(prop,answ(occ))
 WITH
 joint(exec-cont1,exec-cont2): Exec-cont =
          LAM state. LAM input*.
          LET answ1 = exec-cont1(state)(input*)
          LET answ2 = exec-cont2(state)(input*)
          IN
                    LAM occ.
                    compatible(answ1(occ),answ2(occ))
WITH
 update(exp,r-value);exec-cont: Exec-cont =
          LAM state.
          LET state' = state\ exp <- r-value
          IN  exec-cont(state')

 WITH
 content(exp);eval-cont: Exec-cont =
          LAM state.
          LET r-value = state(exp)
          IN  eval-cont(r-value)(state)
```

```
WITH
!IDENTIFIERS
!***********

locate(id)(occ)env;eval-cont:Exec-cont =

          LET [ident] = id IN
           eval-cont(ident)



WITH
!STATEMENTS
!*********

execute(stm)(occ)env;exec-cont :Exec-cont =
          CASE stm
          /[stm+] -> execute-list(stm+)(occ)env;
                     exec-cont

          /["if" exp "then" stml "fi"] ->
                    evaluate(exp)(occ AUG 1)env; LAM t.
                    joint    (execute(stml)(occ AUG 2)(env);exec-cont,
                             exec-cont)

          /["if" exp "then" stml "else" stm2 "fi"] ->
                    evaluate(exp)(occ AUG 1)env; LAM t.
                    joint     (execute(stml)(occ AUG 2)(env);exec-cont,
                                        execute(stm2)(occ AUG 3)(env);exec-cont)


          /["while" exp "do" stm "end"] ->
                    DEF exec-cont'=
                      evaluate(exp)(occ AUG 1)env; LAM t.
                      joint (execute(stm)(occ AUG 2)(env);exec-cont',
                            exec-cont)
                   IN exec-cont'

           /["input" id ] ->
                    locate(id)(occ AUG 1)env; LAM ident.
                    read(ident);LAM r-value.
                    update(ident,r-value);
                    exec-cont

          /["output" exp mode] ->
                    evaluate(exp)(occ AUG 1)env; LAM r-value.
                    exec-cont

          /[id ":=" exp] ->
                     locate(id)(occ AUG 1)env; LAM ident.
                     evaluate(exp)(occ AUG 2)env; LAM r-value.
                     update(ident,r-value);
                     exec-cont

          /["resultis" exp] ->
                    LET eval-cont = env
                    IN evaluate(exp)(occ AUG 1)(env);
                       eval-cont
         ESAC
```

```
WITH
!auxiliary functions for statements
!----------------------------------

execute-list(stm*)(occ)env;exec-cont :Exec-cont =
CASE stm*
/stml PRE stm2* ->
          execute(stml)(occ AUG 1)env;
          execute-list(stm2*)(occ AUG 2)env;
          exec-cont
/<> ->    exec-cont
ESAC
IN
run (prog) (init-state) : (Input*  -> Answ)

END
```

6) CONGRUENCE BETWEEN USUAL INTERPRETATION AND DENOTATIONAL DEFINITION OF PROPERTIES

To establish a connection between the usual semantics and the non-standard one, we must define relations between the two sets of functions. The properties associated with syntactic elements characterized the behavior of semantic objects used in standard interpretation. These objects can be states or continuations, depending of the properties we are interesting in.

6.1) Extension of the standard interpretation

We can explicitely express this behavior by introducing a "trace" of the computations in the domain of answers. This trace is a list of pairs <occ,object> ; and the extended domain of answers including the trace is defined by Answ : <Occ,Object>*

The functions which define this extended standard interpretation are the usual semantic functions modified according to the following pattern :

```
semantic-function (elem)(occ)(env) ; continue : Continuation =
        LAM state.
             LET object be the current semantic object
             IN
             LAM input*.
                  <occ,object>  PRE  continue(state)(input*)
```

The write function is the identity on continuations.

6.2) Relations between domains

We express these relations by predicates defined from pairs :

<Non-standard-domain, Standard-domain>

into T', where T' is a complete lattice having "and" as its join operation, "true" as its least element and "false" as its other element.

Each predicate applied to <⊥,⊥> is made equal to "true".

The exact nature of the predicates will be stated precisely in the application.

Predicates are defined on the semantic functions : their definition is of the form :

P-Elem (property-definition(elem),semantic-definition(elem)) =
(*P-Env* (env-1,env-2) and *P-Cont* (continue-1,continue-2)) $\longrightarrow$ (*)
P-Cont (property-definition(elem)(occ)(env-1);continue-1,
semantic-definition(elem)(occ)(env-2);continue-2)

The standard and non-standard interpretations are congruent if for each syntactic domain Elem, the predicate *P-Elem* is true.

The proof of *P-Elem* is by structural induction on the abstract constructs which belongs to Elem.

6.3) Existence of the used predicates

For a more complex languages, including for example functions with local declarations, the predicates are on recursively defined domains. In such cases their existence is not obvious; they can be constructed as limit of predicates defined on retracts. R. Milne in [9] described such techniques.

6.4) Application

We follow our study of a formal definition of properties associated with expressions.

A property associated with an expression at the occurrence occ is characterized by

- the set of current states each time the control reaches this occurrence

or - the set of current values this expression can take at this occurrence.

Hence the semantic objects we are interested in are states or values depending on the property we define.

(*) a $\longrightarrow$ b denotes : if a is true then b else false.

As we are required to compare pairs of functions (one for each definition scheme) which can be both called by the same name, we will, in a systematic way, add the suffix -1 (respectively -2) to the names belonging to the non-standard interpretations (respectively extended standard one).

Thus let us assume the existence of the following predicate, defined on the basic domains :

P-Prop := <Prop, <Occ,State-2>> → T'
P-Value := <Value-1,Value-2> → T'
P-Input := <Input*-1,Input*-2> → T'

Their exact nature will be stated precisely for each desired property definition described in the next chapter.

We define :

P-Answ := <Answ-1,Answ-2> → T'
by
P-Answ (answ-1,answ-2) =
for any <occ,state-2> which belongs to answ-2
and {*P-Prop* (answ-1(occ), <occ,state-2>)}

P-State := <State-1,State-2> → T'
by
P-State (state-1,state-2) =
for any ident wich belongs to Ident
and {*P-Value* (state-1(ident),state-2(ident))}

We define :

P-Cont := <Continuation-1,Continuation-2> → T'
by
P-Cont (continue-1,continue-2) =
(*P-State* (state-1,state-2) and *P-Input* (input*-1,input*-2)) ⟶ (*)
P-Answ (continue-1(state-1)(input*-1),
continue-2(state-2)(input*-2))

In the same way we define :

P-Eval-cont := <Eval-cont-1,Eval-cont-2> → T'
by
P-Eval-cont (eval-cont-1,eval-cont-2) =
(*P-Value* (value-1,value-2) and *P-State* (state-1,state-2)
and *P-Input* (input*-1,input*-2)) $\xrightarrow{(*)}$
P-Answ (eval-cont-1(value-1)(state-1)(input*-1),
eval-cont-2(value-2)(state-2)(input*-2))

We use this predicate to define *P-Env* :

P-Env = *P-Eval-cont*

In this application we must prove that :

property_1 :
for all exp exp belongs to Exp
P-Exp (evaluate-1(exp),evaluate-2(exp)) is true
where
P-Exp (evaluate-1(exp),evaluate-2(exp) =
(*P-Env*(env-1,env-2) and
P-Eval-cont(eval-cont-1,eval-cont-2)) $\longrightarrow$
P-Cont (evaluate-1(exp)(occ)(env-1)(eval-cont-1),
evaluate-2(exp)(occ)(env-2)(eval-cont-2))

To be able to prove this property we express relations based on the domains and functions which parameterize the function defined in table 4.

relation_1 :

(*P-Value*(value'-1,value'-2) and *P-Value*(value"-1,value"-2)) implies
P-Value(value-1,value-2)
where value = op-val (value', value", op)

relation_2 :

P-Prop(prop, <occ,object>) implies *P-Prop*(prop', <occ,object>)
where prop' is any element of Prop which is compatible with prop

relation_3 :

(*P-State*(state-1,state-2) and *P-Value*(value-1,value-2)) implies
P-Prop(prop, <occ,object>)
where prop = prop-def (value-1)(state-1)

Some technical lemmata are needed to prove property 1.

Lemma_1 :

P-Eval-cont (eval-cont-1,eval-cont-2) implies
P-Cont(content(ident);eval-cont-1,content(ident);eval-cont-2)

Proof

P-State(state-1,state-2) is equal to "true" from the hypothesis.
P-Value(state-1(ident),state-2(ident)) is equal to "true" from the defifinition of *P-State*.

The result is obtained using the hypothesis and the definition of the function content.

Lemma_2 :

(*P-Value*(value-1,value-2) and *P-Cont*(continue-1,eval-cont-2(value-2))) implies
P-Cont(update(exp,value-1);continue-1,eval-cont-2(value-2))

Proof obvious

Lemma_3 :

(*P-Value*(value-1,value-2) and *P-Cont*(continue-1,eval-cont-2(value-2))) implies
P-Cont (attach(occ,value-1);continue-1,
LAM state.LAM input*. <occ,object> PRE eval-cont-2(value-2) (state) (input*))
(where object is used for value-2 or state-2

Proof

P-State (state-1,state-2) and *P-Input* (input*-1,input*-2) are true from the hypothesis.

Let answ-1 = continue-1(state-1) (input*-1)

Let answ-2 = eval-cont-2(value-2)(state-2) (input*-2)

Then *P-Answ*(answ-1,answ-2) is true from its definition

Let answ'-1 = answ-1\occ ← compatible(answ-1(occ),prop)

where prop = prop-def (value-1)(state-1)

The result is obtained using relation 2 and relation 3.

Lemma 4 :

(*P-Cont* (continue'-1,continue'-2) and *P-Cont*(continue"-1,continue"-2)) implie

LET continue-2 = joint (continue'-2,continue"-2)

IN (*P-Cont*(continue-1,continue-2) AND *P-Cont* (continue"-1,continue-2))

the proof is obvious using the definition of *P-Cont* and relation 2.

Assuming the relations defined above are verified we can now prove property 1.

The proof uses a structural induction on the domain Exp.

/ id →

```
evaluate-1(id)(occ)(env-1);eval-cont-1 : Exec-cont-1 =
    content (id) ; LAM value-1.
    attach (occ,value-1);
    update (exp,value-1);
    eval-cont-1 (value-1)

evaluate-2(id)(occ)(env-2);eval-cont-2 : Exec-cont-2 =
    content (id) ; LAM value-2.
    LAM state-2 . LAM input*.
    <occ,object> PRE eval-cont-2(value-2)
```

Using Lemma 1 the result is obtained if

(a) *P-Cont*(attach(occ,value-1);cont-1,cont-2) is true

where cont-1 = update(exp,value-1);
eval-cont-1(value-1)

cont-2 = LAM state-2.
<occ,object> PRE eval-cont-2 (value-2)(state-2)

Using Lemma 3 (a) is proved if

(b) (*P-Cont*(cont-1,eval-cont-2(value-2) and *P-Value*(value-1,value-2)) is true.

From the hypothesis *P-State*(state-1,state-2) is true, hence we know from the definition of the function content that *P-Value*(value-1,value-2) is true

To prove (b) we have only to show that *P-Cont*(cont-1,eval-cont-2(value-2)) is true.

This is done using Lemma 2 and the hypothesis.

/ num, / bool

the proof uses same arguments as in the previous case.

/ "valof" stm

the result is obtained using a same reasoning as in the first case and the induction hypothesis on *P-Stm*.

/ exp1 op exp2

```
evaluate-1(exp1 op exp2)(occ)(env-1);eval-cont-1 : Exec-cont =
   evaluate-1(exp1)(occ AUG 1)(env-1) ; LAM value'-1.
   evaluate-1(exp2)(occ AUG 2)(env-2) ; LAM value"-1.
   LET value-1 = value-def (value'-1,value"-1,op)
   IN attach (occ,value-1);
      update (exp,value-1);
      eval-cont-1 (value-1)

evaluate-2(exp1 op exp2)(occ)(env-2);eval-cont-2 : Exec-cont =
   evaluate-2(exp1)(occ AUG 1)(env-2) ; LAM value'-2.
   evaluate-2(exp2)(occ AUG 2)(env-2) ; LAM value"-2.
   LET value-2 = value-def (value'-2,value"-2,op)
   IN eval-cont-2 (value-2)
```

By hypothesis *P-State*(state-1,state-2) is true.

The induction hypothesis is :

P-Exp(**eva**luate-1(exp),evaluate-2(exp)) = true.

Using the induction hypothesis, the result is obtained if we show that

(a) *P-Eval-cont*(eval-cont'-1,eval-cont'-2) is true

where eval-cont'-1 is the continuation of the function evaluate-1(exp1)(occ)(env-1) in evaluate-1(exp1 op exp2) and eval-cont'-2 is the continuation of evaluate-2(exp1)(occ)(env-2) in evaluate-2(exp1 op exp2)

In other word (a) is true if

(b) [(*P-State*(state'-1,state'-2) and *P-Value*(value'-1,value'-2)) ⟶ *P-Answ*(answ'-1,answ'-2)] is true

where answ'-1 = evaluate-1(exp2)(occAUG2)(env-1)(eval-cont"-1)(state'-1)
answ'-2 = evaluate-2(exp2)(occAUG2)(env-2)(eval-cont"-2)(state'-2)

By induction hypothesis

P-Exp(evaluate-1(exp2),evaluate-2(exp2)) is true

Thus (b) is true if

(c) *P-Eval-cont*(eval-cont"-1,eval-cont"-2) is true

where eval-cont"-1 is the current continuation of evaluate-1(exp2)(occAUG2)(env-1) in evaluate-1(exp1 op exp2) and eval-cont"-2 is the symetric current continuation in evaluate-2(exp1 op exp2)

The predicate (c) is true if

(d) [(*P-State*(state"1,state"2) and *P-Value*(value"-1,value"-2)) ⟶ *P-Answ*(answ"-1,answ"-2)] is true

where answ"-1 = eval-cont"-1 (value"-1)
answ"-2 = eval-cont"-2 (value"-2)

Assume the left part of (d)

From relation 1 we know that *P-Value*(value-1,value-2) is true ; using the same arguments as in the first case we can show that (d) is true ; so if (d) is true the result is proved.

Property 2 :

P-Id(locate-1(id),locate-2(id)) is true.

Trivialy true from the definitions of locate-1 and locate-2.

Property_3 :

P-Stm(execute-1(stm),execute-2(stm)) is true.

The proof uses a structural induction on the domain Stm and a computational induction for the while loop. The result is easily obtain using property 1, property 2 and lemma 4.

Property_4 :

P-Input(input-1,input-2) implies
P-Prog(run-1(prog),run-2(prog))

Proof

assume *P-Input*(input-1,input-2) is true.
P-Env(init-env-1,init-env-2) is true, thus it is enough to show that
P-Cont(execute-1(stm)(init-occ AUG 2)init-env-1;init-cont-1,
execute-2(stm)(init-occ AUG 2)init-env-2;init-cont-2) is true

Using Property 3, it is enough to show that
P-Cont(init-cont-1,init-cont-2) is true.
This is done using the definitions of *P-Cont* and initial continuation.

7) SOME CLASSICAL DATA FLOW ANALYSES

In this chapter we give a denotational definition of some classical data flow analysis which compute properties of expressions.

Each definition is obtained by stating the basic domains and functions which parameterized the definition given in table 4.

For each application, basic predicates are defined and the relations based on the domains and functions are proved. The coherence between usual interpretation and the denotation definition of the properties is shown using property 4.

7.1) Constant propagation

An expression is a constant at an occurrence occ, if in any computation, the value of this expression at occ is constant. The aim is to discover in any program the occurrences of expressions which are known constants.

7.1.1) Basic domains and functions

The domain of values is formed from the basic domain including integer and boolean values plus the element "var". The domain of property is equal to the domain of values. A state associates to each expression a value.

Thus if occ is the occurrence of an expression in a program prog, answ is the result of this non-standard interpretation applied to prog :

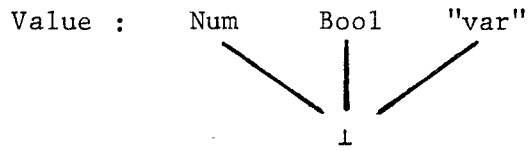
answ(occ) = n if the expression at occ in prog is a constant and its value is n.

answ(occ) = "var" if this expression has not a constant value and

answ(occ) is undefined if this expression has never been evaluated.

The semantic definition described in table 4 parameterized by the domains and functions described in table 5, entirely defines this property.

The function prop-def is the identity on Value. The compatible function defines the compatible operation on the flat domain :



The read function associates to an identifier the value "var".

7.1.2.) Definition of the basic predicates

In this application the semantic objects we consider are values.

We define

P-Value(value-1,value-2)

by

(value-1 EQ "var") or (value-1 EQ value-2)

P-Prop(prop, <occ,value-2>)

by

P-Value(prop,value-2)

P-Input(input*-1,input*-2) is always true since input*-1 equal "var"* and

P-Value(var,input-2) is true

7.1.3) Proof of the relations defined on the basic domains

relation 1 :

(*P-Value*(value'-1,value'-2) and *P-Value*(value"-1,value"-2)) implies

P-Value(value-1,value-2)

where value-1 = op-val - 1 (value'-1,**value**"-1,op)

value-2 = op-val - 2 (value'-2,value"-2,op)

The proof is obvious and uses the definitions of the op-val functions and the predicate *P-Value*.

relation 2 :

if *P-Prop*(prop,<occ,value-2>) is true then

for any prop' compatible with property

P-Prop(prop',<occ,value-2>) is true.

Proof

Let value-1 = prop and value'-1 = prop'.

We have to show that *P-Value*(value'-1,value-2) is true.

If value'-1 is equal to value-1 then the result is true (hypothesis).

If value'-1 is equal to "var" then the result is obtained using the definition of *P-Value*.

relation 3 :

(*P-State*(state-1,state-2) and *P-Value*(value-1,value-2)) implies

P-Prop(prop, <occ,value-2>)

where prop = prop-def (value-1)(state-1)

The proof is obvious.

7.1.4) Coherence problem

In this application *P-Input*(input*-1,input*-2) is trivially true, thus from property 4 we know that

P-Prog(run-1(prog),run-2(prog)) is true.

```
DSL "constant-propagation"
DOMAINS
!SEMANTIC VARIABLES AND DOMAINS
!******************************

state    :State = Exp -> R-value;
r-value  :R-value = N / T /"var";
prop     :Prop = R-value;
exec-cont:Exec-cont = State -> Input* -> Answ;
answ     :Answ;
input    :Input;
eval-cont:Eval-cont = R-value -> Exec-cont;

!TYPES OF FUNCTIONS
!******************

init-prop  := Prop;
wrong      := Exec-cont;
read       := Ident -> Eval-cont -> Exec-cont;
compatible := <Prop,Prop> -> Prop;
prop-def   := R-value -> State -> Prop;
op-val     := <R-value,R-value,Op> -> R-value;

IN
!DESCRIPTION OF FUNCTIONS
!************************

LET   init-prop : Prop = ?


ALSO  read(ident);eval-cont: Exec-cont =
        eval-cont("var")

ALSO  compatible(prop1,prop2): Prop =
         prop1 IS prop2 -> prop1 ,"var"

ALSO prop-def(r-value)(state): Prop =
         r-value

ALSO op-val(r-value1,r-value2,op): R-value =
         (r-value1 IS "var") OR (r-value2 IS "var") -> "var",
         CASE op
         /"+"      -> r-value1 PLUS r-value2
         /"*"      -> r-value1 MULT r-value2
         /"="      -> r-value1 EQ r-value2
         /"#"      -> r-value1 NE r-value2
         /"or"     -> r-value1 OR r-value2
         /"and"    -> r-value1 AND r-value2
         ESAC
IN <init-prop,read,compatible,prop-def,op-val>
END
```

The calculation of an expression is redundant at an occurrence occ, if, in every computation, its value has already been evaluated.

Example [8]

r := a + b ; ... r + x ; ... (a + b) + x ;

Assume that r, a, b and x are not modified.

The second occurrence of r is redundant as well as the second occurrence of (a + b) and the expression (a + b) + x.

This application detects redundant calculations of expressions.

7.2.1) Domains and functions

The result of this non-standard interpretation applied to a program associates with an occurrence occ of an expression exp the set of the expressions of the program that have already been evaluated and have same value at occ than exp.

This set is defined by a predicate from Exp to T' and thus

Prop = Exp → T'

Let answ the result of this non-standard interpretation applied to a program prog :

-if there is an exp' such that

answ(occ)(exp') is true than exp and exp' have same value at occ and the calculation of exp is redundant.

-otherwise the calculation of exp at occ is not redundant.

The basic values of this interpretation are symbolic terms :

-to each constant we associate a symbolic constant identified with the syntactic element

-to each read variable we associate a symbolic value (which is different for each reading). This symbolic value is identified to a syntactic identifier and thus the domain of input values is identified to Ident ie Input = Ident

-to each operator op we associate the symbolic operator "op" which defines from symbolic terms "t_1","t_2" the term "t_1 op t_2".

In fact this domain of symbolic terms is equal to the domain Exp restricted to :

[id]
/ [num]
/ [bool]
/ [exp_1 op exp_2]

Thus a state associates to each expression a symbolic term :

State := Exp → Exp.

This non-standard interpretation is enterely defined by the functions described in table 4 parameterized by the basic domains and functions given in table 6.

Comments on the basic functions.

- The read function associates to an identifier a symbolic value taken from the input stream.
- The compatible operation defined on Prop is the logical operator "and" (ie intersection of sets)
- The prop-def function builds the set of expressions which have same value than its argument.

7.2.2) Definition of the basic predicates

With any element input*-2 of Input*-2 we can associate a valuation function α from the symbolic input stream Input*-1 to Input*-2 defined by

ν(input-1 PRE input*-1) = input-2 PRE ν(input*-1)

thus if id belongs to input*-1, ν(id) is defined as above. We extend this function to the terms in the following way :

ν : Value-1 → Value-2

by

ν(num) = num

ν(bool) = bool

ν(term 1 op term 2) = ν(term1) op ν(term2)

we define :

P-Value(value-1,value-2)

by

ν(value-1) = value-2

In this application the semantic objects considered are states.

We define

P-Prop(prop, <occ,state-2>)
by
for any prog, for any exp", for any input-2
LET exp' = prog at occ
IN prop(exp") ⟶ (value"-2 EQ value'-2)
where
QN*(value'-2)= evaluate-2(exp')(occ)(env-2)(LAM value.LAM state.QN(value))(state-2)
QN (value"-2)= evaluate-2(exp")(occ)(env-2)(LAM value.LAM state.QN(value))(state-2)

7.2.3) Proof of the relations defined on the basic domains

Relation 1 :
(*P-Value*(value'-1,value'-2) and *P-Value*(value"-1,value"-2)) implies
P-Value(value-1,value-2)
where value = op-val(value',value",op)

The proof is obvious using the definition of the valuation function ν, the definition of *P-Value* and the definitions of the value-def functions.

Relation 2 :
if *P-Prop*(prop, <occ,state-2>) is true then
for any prop' such that prop and prop' are compatible :
P-Prop(prop', <occ,state-2>) is true

The proof is obvious since in this application the compatible operation is the intersection (i.e. "and" operation on T')

Lemma 8 :
for any input-2
P-State(state-1,state-2) implies
P-Value(state-1(exp),value-2)
where
QN(value-2)= evaluate-2(exp)(occ)(env)(LAM value.LAM state.QN(value))(state-2)(input-2)

The proof is done by structural induction on exp.

Relation 3 :
for any prog, for any occ occ ∈ Occ(prog), for any input-2
(*P-State*(state-1,state-2) and *P-Value*(value-1,value-2)) implies
P-Prop(prop, <occ,state-2>)
where
prop = prop-def (value-1)(state-1)
Qn(value-2)= evaluate-2(prog at occ)(occ)(env)(LAM value.LAM state.QN(value))(state-2)(input-2)

---

* QN stands for the SIS function.LAM value.QUOTE(NUMBER(value))

Proof

LET exp = prog at occ

From the definition of the function prop-def we know that for any exp' :

if prop(exp') = true then state-1(exp') = value-1

From Lemma 8 we know that

ν(state-1(exp')) = value'-2

where

value'-2 = evaluate-2 (exp')(occ)(env)(LAM value.LAM state.QN(value))(state-2)
(input-2)

Using the definition of *P-Value* we can write

ν(state-1(exp)) = value-2

hence

ν(state-1(exp')) = value-2

ν(value-1) = value-2

and value'-2 = value-2

Table 6

```
DSL "common subexpressions"
DOMAINS

!SEMANTIC VARIABLES AND DOMAINS
!******************************

r-value :R-value = Exp;
state   :State   = Exp -> R-value;
prop    :Prop    = Exp -> T;
exec-cont:Exec-cont = State -> Input* -> Answ;
answ    :Answ;
eval-cont:Eval-cont = R-value -> Exec-cont;
input   :Input   = Q;

!TYPES OF FUNCTIONS
!******************

init-prop := Prop;
read      := Ident -> Eval-cont -> Exec-cont;
compatible:= <Prop,Prop> -> Prop;
prop-def  := R-value -> State ->Prop;
op-val    := <R-value,R-value,Op> -> R-value;
wrong     := Exec-cont;

IN
!DESCRIPTION OF FUNCTIONS
!************************

LET   init-prop :Prop = LAM exp. FF

ALSO read(ident);eval-cont: Exec-cont =
         LAM state.
         LAM input*.
         CASE input*
         /input1 PRE input2* ->
                  LET exp = [input1]
                  IN eval-cont(exp)(state)(input2*)
         /<>     -> wrong(state)(input*)
         ESAC

ALSO     compatible(prop1,prop2): Prop =
                           LAM exp.
                            prop1(exp) AND prop2(exp)

ALSO     prop-def(r-value)(state): Prop =
                            LAM exp.
                            state(exp) IS r-value -> TT,FF


ALSO     op-val(r-value1,r-value2,op): R-value =
                  CASE op
                  /"+" ->   [r-value1 "+" r-value2]
                  /"*" ->   [r-value1 "*" r-value2]
                  /"=" ->   [r-value1 "=" r-value2]
                  /"#" ->   [r-value1 "#" r-value2]
                  /"or"->   [r-value1 "or" r-value2]
                  /"and" -> [r-value1 "and" r-value2]
                  ESAC

IN <init-prop,read,compatible,prop-def,op-val>
END
```

7.3) Determination of invariant expressions

An expression is invariant at the occurrence occ of a program prog if its value remains constant each time the control reaches occ.

We want to determine the expressions that are invariant in any usual interpretation.

This property can be defined by an interpretation similar to the one used for the constant propagation problem defined with a basic value domain of symbolic terms.

7.3.1) Domains and functions

Thus the domain of values is equal to the domain of symbolic terms ie Exp plus the element "var".

Value = Exp / "var".

A state associates with each expression a value.

If occ is an occurrence, answ is the result of this non-standard interpretation :

answ(occ) = exp if the expression at occ is invariant and exp is its symbolic value.

answ(occ) = "var" if not

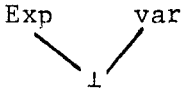
if answ(occ) is undefined then this expression has never been evaluated.

This interpretation is enterely defined by the table 4 and the table 7.

An input is identified with a syntactic identifier.

The read function associates with an identifier a symbolic value taken from the input stream.

The compatible function defines the compatible operation on the flat domain :



7.3.2) Definition of the used predicates

As in the precedent application, we build a valuation function ν from the symbolic and standard inputs.

Using this function we define :

P-Value(value-1,value-2) =

(ν(value-1) EQ value-2) OR (value-1 EQ "var"))

In this application, semantic objects are values.

We define

P-Prop(prop,<occ,value-2>)

by

P-Value(value-1,value-2)

where

value-1 = prop

P-Input is true from the definition of ν.

7.3.3) Proof of the basic relations

relation 1 :

(*P-Value*(value'-1,value'-2) and *P-Value*(value"-1,value"-2)) implies

P-Value(value-1,value-2)

where

value = op-val (value' ,value" ,op)

The proof is obvious from the definition of ν, *P-Value* and the functions value-def.

relation 2 :

If *P-Prop*(prop,<occ,value-2>) is true then

for any prop' compatible with prop

P-Prop(prop',<occ,value-2>) is true

The proof of this relation is the same as in the case of the constant propagation problem.

relation 3 :

for any prog, for any occ such that occ ∈ Occ(prog)

P-State(state-1,state-2) and *P-Value*(value-1,value-2)) implies

P-Prop(prop, <occ,value-2>)

where

prop = prop-def (value-1,state-1)

The proof uses the definition of the function ν and follows the same reasoning as in the constant propagation problem.

7.3.3) Coherence problem

As *P-Input*(input-1,input-2) is true, this problem is solved using property 4. Hence

P-Prog(run-1(prog),run-2(prog)) is true.

Table 7

```
DSL "invariant expressions"
DOMAINS
!SEMANTIC VARIABLES AND DOMAINS
!******************************

state     :State  = Exp -> R-value;
r-value   :R-value= Exp / "var";
prop      :Prop   = R-value;
exec-cont:Exec-cont= State -> Input* -> Answ;
answ      :Answ;
input     :Input  = Q;
eval-cont:Eval-cont= R-value -> Exec-cont;

!TYPES OF FUNCTIONS
!******************

init-prop  := Prop;
read       := Ident -> Eval-cont -> Exec-cont;
compatible := <Prop,Prop> -> Prop;
prop-def   := R-value -> State -> Prop;
op-val     := <R-value,R-value,Op> -> R-value;
wrong      := Exec-cont;

!DEFINITIONS OF FUNCTIONS
!************************

LET init-prop: Prop  =?
ALSO read(ident);eval-cont: Exec-cont =
        LAM state.
         LAM input*.
         CASE input*
         /input1 PRE input2* ->
                  LET exp = [input1]
                  IN eval-cont(exp)(state)(input2*)
         /<>  ->   wrong(state)(input*)
         ESAC

ALSO  compatible(prop1,prop2): Prop =
                   prop1 IS prop2 -> prop1,["var"]

ALSO  prop-def(r-value)(state): Prop =
                   r-value

ALSO  op-val(r-value1,r-value2,op): R-value =
                   (r-value1 IS ["var"]) OR (r-value2 IS ["var"]) -> ["var"],
                  CASE op
                   /"+" ->   [r-value1 "+" r-value2]
                   /"*" ->   [r-value1 "*" r-value2]
                   /"=" ->   [r-value1 "=" r-value2]
                   /"#" ->   [r-value1 "#" r-value2]
                   /"or"  ->  [r-value1 "or" r-value2]
                   /"and" -> [r-value1 "and" r-value2]
                   ESAC

IN
<init-prop,read,compatible,prop-def,op-val>
END
```

ACKNOLEDGEMENTS :

I thank R. Burstall and R. Tennent for their interest in this study and for their useful comments, P. Mosses for his help in using his system SIS, B. Lang, G. Kahn and G. Huet for their continuous support.

REFERENCES

[1] *Aho A.V. and Ullman J.D.*
"The theory of Parsing, Translation and Compiling"
vol 11, Prentice Hall, 1973.

[2] *Cousot P. and Cousot R.*
"Abstract interpretation : a unified lattice model for static analysis of programs by construction of approximation of fixpoints"
Proc. ACM Symp. on Principles of Programming Languages 1977, (238-252).

[3] *Donzeau-Gouge V.*
"Utilisation de la sémantique dénotationnelle pour la description d'interprétations non-standard : application à la validation et à l'optimisation des programmes"
Third International Symposium on Programming. Dunod 1978.

[4] *Donzeau-Gouge V., Kahn G., Lang B.*
"A complete machine-checked definition of a simple programming language using denotational semantics"
Rapport Laboria n° 330

[5] *Fong A., Kam J., Ullamn J.*
"Application of lattice algebra to loop optimization"
Proc. ACM Symp. on Principles of Programming Languages 1975, (1-9).

[6] *Hecht M-S.*
"Flow analysis of computer programs"
North-Holland-New York.

[7] *Kam J.B., Ullman J.D.*
"Global data flow analysis and iterative algorithms"
J. A. C. M. vol 23 n°1, 1976 (158-171).

[8] *Kildall G.A.*
"A unified approach to global program optimization"
Proc. ACM Symp. on Principles of Programming Languages 1973, (194-206).

[9] *Milne R. and Strachey C.*
"A theory of programming language semantics"
Chapman and Hall.

[10] *Mosses P.D.*
"SIS : a compiler-generator system using denotational semantics"
DAIMI, University of Aarhus, 1978.

[11] *Mosses P.D.*
"The mathematical semantics of Algol 60"
Technical monograph PRG-12, Programming Research Group, Oxford.

[12] *Mosses P.D.*
"Mathematical semantics and compiler generation"
D. Phil. Thesis, University of Oxford, 1975

[13] *Scott D.*
"Notes de cours"
Séminaire avancé de sémantique. Sophia Antipolis, Septembre 1977.

[14] *Sintzoff M.*
"Calculating properties of programs by valuations on specific models"
Proc. ACM conference on proving assertions about programs,
New Mexico, 1972, (203-207).

[15] *Stoy J.E.*
"Denotational Semantics : The Scott Strachey approach to programming language theory"
MIT Press 1977.

[16] *Tennent R.D.*
"The denotational semantics of programming languages"
Comm. ACM 19, 1976, (437-453).

[17] *Tennent R.D.*
"A denotational definition of the programming language Pascal"
To be published.

[18] *Wegbreit B.*
"Property extraction in well founded property sets"
IEEE transactions on software engineering
vol SE 1 N°3 September 1975, (270-285).