SEMANTICS IMPLEMENTATION SYSTEMS

(Extended Abstract)

P. Mosses

Rapporteur: Mr. R.C. Millichamp

Abstract

In this talk, I start by briefly outlining the main ways of specifying programs, and consider the automatic implementation of these specifications. Denotational Semantics is introduced as a means of specifying compilers (or interpreters) of programming languages. A particular system, SIS, for implementing denotational specifications is described, and its history and uses are reviewed.

Formal Specifications of Programs

Here the aim is only to distinguish the main approaches to the specification of (non-concurrent) programs.

Consider programs that, when executed, read some input and write some output - perhaps without terminating. A specification of such a program describes some relevant (to its users and implementors) details, such as its input-output (I-O) relation, its termination domain, or, less commonly, its required modular structure, its efficiency, etc. Let us consider specifying a program's I-O relation.

One way of specifying the I-O relation of a program is to state that it should be the same as that of some given program. Such a specification may be called operational: the I-O relation is given, indirectly, by operation of a machine implementing the given program. This machine might be an ideal, abstract machine.

Another way of giving information about the I-O relation of a program is by means of pre- and post-conditions: when the input satisfies the precondition, the output must satisfy the postcondition (on termination, usually). This may be regarded as an axiomatic form of specification (assertional might be a better word): the conditions do not indicate how the output is determined by the input, (in fact there might not exist a program computing the specified I-O relation!).

Finally (in this brief sketch) one might use mathematical, functional notation to specify an I-O relation. Like an operational specification, this shows how the output is to depend on the input, but the use of functional notation here gives a more direct specification of this, as there is no need to consider the operation of an implementing machine.

Automatic Implementation of Formal Specifications

First, what are the uses of automatic implementations? I can see two main ones: checking specifications, both for internal consistency (e.g. syntactic checks) and against the intentions of the specifier; and in providing an initial implementation of the specified program.

Regarding the checking of specifications against intentions, the aim is merely to expose gross faults in the specification, rather than to convince anyone that the specification really does express these intentions.

In general, automatic implementations are many orders of magnitude less efficient than hand-programmed implementations (accounted in computer time rather than implementation time, anyway!). For some particular programs, though, such as parsers, automatic implementations of specifications may even have superior efficiency.

There are two main sorts of automatic implementation: interpreters and compilers (with specifications now playing the part of programs). An interpreter takes both a specification of a program and its input, and produces the output, whereas a compiler produces some code from a specification, and this code can later be run with different inputs to produce the corresponding outputs.

Conventional compiling and interpreting techniques can be used to give automatic implementations of operational specifications. For a special form of axiomatic specification, the PROLOG system provides an interpretive implementation. We shall be looking more closely at SIS, which is a system for implementing functional specifications, in particular Denotational Semantics.

Denotational Semantics

Rather than introducing Denotational Semantics as a means of specifying programming languages, per se, let us regard it as a tool for specifying compilers (or interpreters) of programming languages.

A concrete compiler takes source programs as input, and produces target code as output - this code itself is really an (operational) specification of the I-O relation of the source program. Abstracting the choice of a particular target language, we get the notion of an abstract compiler whose output is any representation of the source program's I-O relation. Alternatively, we may consider an interpreter which, when given both the source program and the input, produces the output.

EXAMPLE : LOOP

Abstract Syntax :

(Prog) prog ::= read var; cmd; write exp

(Cmd) cmd ::= cmd_1; cmd_2

| var := exp

| to exp do cmd

| (cmd)

(Exp) exp ::= 0

| var

| succ exp

(Var) var -- standard

Semantic Values :

s $\in$ S = Var $\to$ N

n $\in$ N -- standard natural numbers

Semantic Functions

$\mathcal{P}$: Prog $\to$ (N $\to$ N)

$\mathcal{C}$: Cmd $\to$ (S $\to$ S)

$\mathcal{E}$: Exp $\to$ (S $\to$ N)

$\mathcal{P}[\![$ read var; cmd; write exp $]\!](n) = \mathcal{E}[\![exp]\!](\mathcal{C}[\![cmd]\!]((\lambda var'.0)[n/var]))$

$\mathcal{C}[\![cmd_1; cmd_2]\!](S) = \mathcal{C}[\![cmd_2]\!](\mathcal{C}[\![cmd_1]\!](S))$

$\mathcal{C}[\![var := exp]\!](S) = S[\mathcal{E}[\![exp]\!](S)/var]$

$\mathcal{C}[\![$ to exp do cmd $]\!](S) = (\mathcal{C}[\![cmd]\!])^{\mathcal{E}[\![exp]\!](S)}(S)$

$\mathcal{E}[\![0]\!](S) = 0$

$\mathcal{E}[\![var]\!](S) = S(var)$

$\mathcal{E}[\![$succ exp$]\!](S) = \mathcal{E}[\![exp]\!](S) + 1$

Note: $(S[n/var])(var) = \begin{cases} n, & \text{if } var = var \\ S(var), & \text{otherwise} \end{cases}$

A denotational Semantics specification can be regarded as a functional specification of an abstract compiler*: no choice of target code is made, but a function from programs to their I-O relations is specified. Moreover, it is required that the specification gives denotations not only to complete programs (their I-O relations) but also to arbitrary phrases of programs, with the denotation of a compound phrase being composed from the denotations of that phrase's components.

SIS, A Semantics Implementation System

SIS provides automatic implementation of Denotational Semantics: given a Denotational specification, a program in the specified source language and the input to that program, SIS can produce the output of the program (it can produce finite approximations to the outputs of non-terminating programs).

It is also possible to get SIS to produce intermediate results, giving it the characteristics of a Compiler-Compiler - given a Denotational specification, it produces (a representation of) the specified abstract compiler. In fact SIS enables the abstract compiler to be used as an interpreter, thus avoiding the production of (a representation of) a program's I-O relation - so SIS is also an Interpreter-Compiler!

The basic idea behind the operation of SIS is to represent all objects in λ-notation: syntactic objects are represented by λ-expressions denoting trees, whereas the object that they specify are represented by λ-expressions denoting functions. After converting a Denotational specification into the (semantic) function it denotes (meta-compilation), SIS works entirely by making formal applications of λ-expressions to each other. This trivially gives representations of the desired objects, but it is not until SIS simplifies these complex λ-expressions to so-called normal form (no more simplification possible) that one can apprehend these objects directly.

Incidentally, meta-compilation can be performed by applying a λ-expression representing the semantic function for the (functional) specification language itself, to specification (tree)s. When this is done for the meta-circular specification of the specification language in itself, this should (and does) yield the same λ-expression again, up to renamings anyway.

Concerning the way that SIS simplifies λ-expressions, the (β-) reduction of $(\lambda x.\sim x\sim x\sim)\ (e)$ to $\sim e\sim e\sim$ is accomplished by simulated substitution via closures, with "call-by-need" providing a safe and practically optimal strategy for choosing the order of reductions. To allow the compilation of programs with loops or recussion to terminate, SIS doesn't unwind applications of Curry's "Paradoxical Combinator" $Y(\sim)$, unless the result could lead to a simplification.

* ignoring the parsing aspect of compiling!

The Basic Idea of SIS :

- Meta-compilation :

```
spec-tree        ⟶        spec-sem.fn.
   (λ)                        (λ)
```

- Compilation:

```
(sem.fn.)(prog-tree) ⇒ prog-I/O-fn
   (λ)       (λ)          (λ)
```

- Execution :

```
(I/O-fn)(input)   ⇒     output
   (λ)    (λ)             (λ)
```

- Interpretation :

```
(sem.fn)(prog-tree)(input) ⇒
   (λ)       (λ)      (λ)
                    (output)
                      (λ)
```

- Meta-compilation (λ) :

```
(meta-sem.fn.)(spec-tree) ⇒
    (λ)           (λ)
                (spec-sem.fn.
                   (λ)
```

Uses of a Semantics Implementation System

A SIS could be of use in connection with language design and standards, if formal semantic specifications are used to document design decisions and impose uniformity on implementations. In particular, a SIS could help check the internal consistency of a semantic description, as well as executing test programs.

A SIS might also provide easy, portable and correct implementations of minor languages from their semantic descriptions - although these would be painfully slow with the present state-of-the-art.

Finally, a SIS can provide the same sort of direct feedback for students learning about semantics specifications, that they get from compilers when learning about programming (languages).

The author's SIS itself might be considered good for its generality, the direct acceptance of ordinary Denotational descriptions and its (BCPL-based) portability. Unfortunately: it is terribly inefficient, even on medium-size examples; one has to worry about the concrete syntax of the source language; and the semantic notation used is not identical to that used in any book on Denotational Semantics.

AN EXAMPLE OF THE USE OF SIS

```
.TYPE SEMFNS.TXT

DSL      "LOOP"

DOMAINS ! Abstract Syntax:

     prog:     Prog      =        ["READ" Var ";" Cmd ";" "WRITE" Exp ]
                         ;
     cmd:      Cmd       =        [Cmd ";" Cmd]
                         /        [Var ":=" Exp]
                         /        ["TO" Exp "DO" Cmd]
                         /        ["(" Cmd ")"]
                         ;
     exp:      Exp       =        ["0"]
                         /        [Var]
                         /        ["SUCC" Exp]
                         ;
     var:      Var       =         Q
                         ;

DOMAINS ! Semantic Values:

     c:        C         =        S → S ;

     s:        S         =        Var → N ;

     n:        N         ;

     q:        Q         :
```

```
.TYPE PARSER.TXT

GRAM    "LOOP"

SYNTAX

     Prog     ::=       "READ" var ";" cmd-list ";" "WRITE" exp
                ;
     cmd-list::=       cmd-list ";" cmd
                /      cmd                        : cmd
                ;
     cmd      ::=       var ":=" exp
                /      "TO" exp "DO" cmd
                /      "(" cmd-list ")"
                ;
     exp      ::=       "0"
                /      var
                /      "SUCC" exp
                ;
     var      ::=       "VAR" q                    : q
                ;

DOMAINS

     prog:              Prog ;
     cmd,cmd-list:      Cmd ;
     exp:               Exp ;
     var:               Var ;

LEXIS

     prog     ::=       word+                      : CONC word+
                ;
     word     ::=       var                        : <OUT"VAR", var>
                /      layout                     : <>
                ;
     var      ===       "a"..."z"
                ;
     layout   ===       " " / CC"C" / CC"L" / CC"T"
                ;
END
```

```
DOMAINS ! Semantic Functions:
        mm:=            Prog -> N -> N ;
        cc:=            Cmd  -> S -> S ;
        ee:=            Exp  -> S -> N ;
DEF     mm  [ "READ" var ";" cmd ";" "WRITE" exp ]  (n) : N =
        ee (exp) ( cc (cmd) ( (LAM var'.0)   var  <- n ) )
WITH    cc (cmd') (s) : S =
CASE    cmd'
/         [ cmd1 ";" cmd2 ] ->     cc (cmd2) ( cc (cmd1) (s) )
/         [ var ":=" exp  ] ->     s   var <- (ee (exp) (s))
/         ["TO" exp "DO" cmd ] -> repeat (ee (exp) (s)) (cc (cmd)) (s)
/         ["(" cmd ")" ] ->        cc (cmd) (s)
ESAC
WITH    ee (exp') (s) : N =
CASE    exp'
/         ["0" ] -                 0
/         [var ] -                 s (var)
/         ["SUCC" exp ] -          (ee (exp) (s)) PLUS 1
ESAC
WITH    repeat (n) (c) (s) : S =
        n EQ 0 ->       s ,
        repeat (n MINUS 1) (c) (c (s))
IN      mm
END
```

```
.TYPE LOOP.LOG
13:27:19 (RUN: 0 sec)      SIS    Version 1.2    1982-9-1
13:27:19 (RUN: 1 sec)      parse("PARSER", "GRAMPR", 0)
13:27:46 (RUN: 19 sec)     PASS 1 finished
13:28:4  (RUN: 32 sec)     PASS 2 finished
13:28:7  (RUN: 33 sec)     ... finished
13:28:7  (RUN: 34 sec)     gram("PARSER", 0, 0)
13:28:32 (RUN: 51 sec)     ...finished
13:28:32 (RUN: 52 sec)     parse("SEMFNS", "DSLPR", 0)
13:29:27 (RUN: 91 sec)     PASS 1 finished
13:30:18 (RUN: 124 sec)    PASS 2 finished
13:30:24 (RUN: 128 sec)    ... finished
13:30:25 (RUN: 129 sec)    dsl("SEMFNS")
13:30:33 (RUN: 135 sec)    ... finished

.TYPE PROG.LOG
13:33:53 (RUN: 0 sec)      SIS    Version 1.2    1982-9-1
13:33:54 (RUN: 1 sec)      parse("PROG", "PARSER", 0)
13:34:3  (RUN: 6 sec)      PASS 1 finished
13:34:5  (RUN: 8 sec)      PASS 2 finished
13:34:6  (RUN: 9 sec)      ... finished
13:34:7  (RUN: 9 sec)      parse("INPUT", "DSLPR", 0)
13:34:30 (RUN: 22 sec)     PASS 1 finished
13:35:0  (RUN: 43 sec)     PASS 2 finished
13:35:2  (RUN: 43 sec)     ... finished
13:35:3  (RUN: 43 sec)     dsl("INPUT")
13:35:4  (RUN: 44 sec)     ... finished
13:35:5  (RUN: 45 sec)     interpret("PROG", "SEMFNS", "INPUT", 9999)
13:37:30 (RUN: 156 sec)    (8654 cycles)
13:37:35 (RUN: 156 sec)    ... finished
13:37:35 (RUN: 157 sec)    write(lamb, "PROG", "OUTPUT", 9999, 79)
13:37:36 (RUN: 157 sec)    ... finished

.TYPE PROG.TXT

READ x;
y := 0;
TO x DO
( TO x DO y := SUCC y );
WRITE y

.TYPE INPUT.TXT

DSL "Input" 7 END

.TYPE OUTPUT.
LAMB "LOOP(PROG)(Input)"

49

END

.
```

LITERATURE

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

M.J.C. Gordon:
"The denotational description of programming languages", (Springer, 1979).

P.D. Mosses:
"Compiler generation using denotational semantics", in Proc. MFCS '76, Gdansk, LNCS (Springer).

"SIS-Semantics Implementation System: Reference Manual and User Guide",
DAIMI, MD-30, Computer Science Dept., Aarhus Univ. (August, 1979).

L. Paulson:
"A semantics-directed compiler generator", in : Proc. ACM Conf. on Principles of Programming Languages, 1982.

J. Bodwin et al.:
"Experience with an experimental compiler generator based on denotational semantics",
in : Proc. 2nd ACM SIGPLAN Symp. on Compiler Construction, Boston, June, 1982.

DISCUSSION

Professor Mosses was asked what the advantages of denotational semantics were over VDL. He explained that he thought Cliff Jones was better qualified to answer and passed the question to him. In reply **Professor Jones** explained that proofs in denotational semantics were easier because operational semantics often included features which got in the way of proofs and consequently made them more difficult.