Beauty and the Beast
or
A Formal Semantic Description of the Control Constructs of Cobol and its Implementation

(IFI TR 93.39)

Peter Baumann, Jürg Fässler, Markus Kiser, Zafer Öztürk *
Institut für Informatik der Universität Zürich,
Winterthurerstr. 190, 8057 Zürich, Switzerland
Email: {baumann,faessler,kiser,oeztuerk}@ifi.unizh.ch

Abstract

We present a formal semantical analysis of the central control constructs of Cobol. In this paper we provide the denotational semantics of a language we call *MiCo*[1] containing the essential control constructs of Cobol. *MiCo* can therefore be regarded as a core language of Cobol. The semantics we have developed has been implemented in the functional programming language Standard ML, thereby resulting in an interpreter for *MiCo*. It can easily be adapted to formal semantics of essential parts of different Cobol dialects.

1 Introduction

In this paper we provide a formal semantical analysis of the central control constructs of Cobol. The theoretical framework we use is *denotational semantics*. We are mainly interested in modelling the meaning of the essential control constructs such as goto-statements, perform-statements, paragraphs, etc.

There is already a considerable literature on denotational semantics (see e.g. [5, 8]), but to the best of our knowledge no such formal analysis of the control constructs of Cobol has yet been published. There exist descriptions of goto-statements (e.g. [2, 8]), but as

* *Jürg Fässler* is supported by KWF (Schweizerische Komission zur Förderung der wissenschaftlichen Forschung), grant nr. 2326.1 *Zafer Öztürk* is supported by Swiss Life Insurance and Pension Company.

[1] *MiCo* stands for **Mi***ni* **Co***bol*

we will see later the meaning of goto-statements in Cobol is not a standard one, which is due to the interaction with the perform-statements.

It may be uncommon to try to develop a formal semantics for Cobol. But one has to realize that of the huge number of real-world Cobol programs there is only a small fraction that will ever be completely rewritten or ported to another language. Most of the existing Cobol programs in use are subject to enormous maintenance efforts. As is well-known, software maintenance is costly and not at all easy.

Our work is motivated by a project on software reverse engineering. In this project (called *AEMES*[2]) we develop a software maintenance environment for Cobol 74 programs. Cobol is well-known for its unstructured control constructs. There are two reasons for us to use formal description techniques. Firstly, in our opinion it is indispensable to really understand the control constructs of Cobol in order to be able to do sound program analysis and semantic invariant program transformations. Secondly, there exist many powerful analysis and meta programming techniques (e.g. abstract interpretation, partial evaluation, continuation passing style based techniques [3, 6, and references therein]) which have as a prerequisite a formal description of the semantics of a programming language. A first step in this direction is done in this paper by describing the semantics of the essential control constructs of Cobol formally. It is worth noting that it is not our goal to develop a full formal semantics of Cobol. We simply want to have a formal semantics of the parts which are essential for further formal analysis. It is definitely a challenge to find out how helpful such advanced techniques can be in the complex task of re-engineering and reverse engineering.

For reasons of space we have chosen to present the semantics of a language we call *MiCo*. This language can be regarded as a core language for Cobol 74 [1] in that it was designed with the goal to include the essential control constructs of Cobol 74.

Our paper has the following structure. In the next section we describe *MiCo* informally and compare it to Cobol 74. The third section explains informally with the help of a small example program written in *MiCo* the interplay between perform-statements and paragraphs. The fourth section contains and explains the formal semantics of *MiCo*. We

[2] *AEMES* stands for "**A**n **E**xtensible **M**aintenance **E**ngineering **S**ystem"

conclude this paper with some remarks and an out-look to our future work in the last section. An appendix contains parts of the denotational interpreter written in Standard ML which are needed for explanation purposes in the fourth section.

We assume some familiarity with continuation style semantics, and Standard ML. For good introductions to these subjects we refer to [2, 4, 5, 7, 8].

2 *MiCo* versus Cobol 74

The development of *MiCo* was triggered by the desire to have a small programming language at hand which makes it possible to describe and analyze the control constructs of Cobol in a concise way. Therefore, we excluded any statement of Cobol with no influence on the control flow. The two exceptions are the skip-statement and move-statement since we wanted to be able to write working example programs. We also did not include the call-statement of Cobol as this is similar to procedure calling in other languages which is well-understood from a formal semantical point of view [2].

The control features we concentrate on are the different perform-statements and the goto-statement of Cobol. As will be seen in the fourth section there is an interesting interplay between the perform-statements, the goto-statement and the meaning of paragraphs. We have not included sections of Cobol in *MiCo* as their meaning is similar to the meaning of paragraphs.

We would like to emphasize that the abstract syntax tree of our language *MiCo* and the one we developed for Cobol 74 in our project are very similar. This is due to the fact that we have a two-level approach for Cobol where in a first step we are not interested in parsing the statements with no influence on the control flow. These statements are simply represented as lists of tokens at the leaves in the syntax tree. This means that based on the formal description of the control constructs of *MiCo* it is easy to get a formal semantics of the control constructs of Cobol 74. This also makes it easy to adapt our description to other Cobol dialects. It is only in a second step that we perform an "on demand fuzzy parsing" of the remaining statements of Cobol.

All this motivates the concrete grammar for *MiCo* which is presented in figure 1.

```
Program            →  PROGRAM Identifier
                        DATA DIVISION DeclarationSeq END
                        PROCEDURE DIVISION ParagraphSeq END
                      END
DeclarationSeq     →  Declaration { Declaration }
Declaration        →  VariableSeq ":" Type
VariableSeq        →  Variable { "," Variable }
Variable           →  Identifier { ALIAS Identifier }
Type               →  NUMERIC
ParagraphSeq       →  Paragraph { Paragraph }
Paragraph          →  PARAGRAPH Identifier StatementSeq END
StatementSeq       →  Statement { "." Statement }
Statement          →  MoveStatement | IfStatement
                   |  GotoStatement | SkipStatement
                   |  PerformStatement
SkipStatement      →  SKIP
MoveStatement      →  MOVE AExpression TO Identifier
GotoStatement      →  GOTO Identifier
PerformStatement   →  PERFORM Identifier [ THRU Identifier ]
                        [ UntilClause | TimesClause ]
UntilClause        →  UNTIL BExpression
TimesClause        →  TIMES AExpression
IfStatement        →  IF BExpression THEN StatementSeq
                        ELSE StatementSeq END
BExpression        →  BTerm { OR BTerm }
BTerm              →  BFactor { AND BFactor }
BFactor            →  NOT BFactor | "(" BExpression ")"
                   |  AExpression ( "=" | "<" | ">" ) AExpression
AExpression        →  ATerm { ( "+" | "-" ) ATerm }
ATerm              →  AFactor { ( "*" | "/" ) AFactor }
AFactor            →  Number | Identifier
                   |  ( "+" | "-" ) AFactor | "(" AExpression ")"
Identifier         →  unspecified
Number             →  unspecified
```

Figure 1: EBNF of *MiCo* (Terminal symbols in uppercase or in quotation marks)

3 Informal Semantics

In this section we analyze informally a small example program *Factorial-of-6* written in *MiCo*. The goal is to explain the interaction between perform-statements and paragraphs. This informal description should make it easier to understand the formal semantics in the fourth section. The example program *Factorial-of-6* is listed in figure 2.

```
PROGRAM Factorial-of-6

  DATA DIVISION n, acc : NUMERIC END

  PROCEDURE DIVISION
    PARAGRAPH Initialization
      MOVE 6 TO n .
      MOVE 1 TO acc END

    PARAGRAPH Control PERFORM Computation UNTIL NOT n > 2 END

    PARAGRAPH Computation
      MOVE acc * n TO acc .
      MOVE n - 1 TO n  END
  END
END
```

Figure 2: Factorial of 6

The program *Factorial-of-6* is supposed to compute the factorial of 6. Thanks to our interpreter based on the denotational semantics for *MiCo* we can confirm that the value of the variable `acc` after the execution of *Factorial-of-6* is 720.

How does *Factorial-of-6* work? The program is divided into two divisions. The data division contains declarations of the variables `n` and `acc`. The second division, called procedure division, is divided into paragraphs. Each paragraph accomplishes a specific task. In the paragraph `Initialization` the two variables are initialized with the help of move-statements which are, in fact, nothing else then assignments. The idea is that `acc` contains at the end the result of the whole computation. The next paragraph `Control`, using a perform-statement, takes us through the repeated execution of the paragraph `Computation` until the condition `NOT n > 2` becomes true, i.e. the paragraph `Computation` is repeatedly executed until the variable `n` has value 2.

The flow of control is as follows. First the condition `NOT n > 2` is checked. If it is false control is switched to the paragraph `Computation`. After reaching the keyword `END` of the paragraph `Computation` control returns to the perform-statement. If the condition `NOT n > 2` is true then the flow of control goes to the lexically following syntactic object after the perform-statement. In our case this means that the control reaches the next paragraph (paragraph `Computation`) which is then executed one more time. In summary, immediately after the perform-statement the variable `acc` contains the value 360 which is equal to the result of calculating $(6 * 5 * 4 * 3)$ and the variable `n` contains the value 2. Then the paragraph `Computation` is executed a last time yielding for `acc` the value 720 and for `n` the value 1 and the execution ends.

We now focus our attention to the keyword `END` of the paragraph `Computation`. On the one hand the flow of control after `END` is determined dynamically. This is the case when the paragraph `Computation` is executed based on a perform-statement. If called from different perform-statements the flow of control is determined dynamically by the calling different perform-statement. In this respect a paragraph like `Computation` can be seen as a procedure without parameters which can be called from different places.

On the other hand the flow of control is sometimes also determined by the syntactical ordering of the paragraphs. This is the case when flow of control reaches a paragraph not through a perform-statement. From this point of view a paragraph is just a sequence of statements.

These observations are the reason for the form of formal semantics we present in the next section. As continuations are used to model the flow of control in denotational semantics it is clear by now that we have to do some static as well as dynamic bookkeeping for continuations of paragraphs.

4 Formal Semantics

4.1 Abstract Syntax

We start our formal semantic definition by giving an abstract syntax of the programming language *MiCo*. As we have chosen to implement our semantics in Standard ML, this syntax is presented in form of a Standard ML datatype `PROGRAM_AST` in figure 3. This definition is easily understandable if one keeps in mind that Standard ML does not allow forward declarations. Therefore, the definitions appear bottom up with the definition of `PROGRAM_AST` at the end. We have omitted the exact definition of the arithmetic expressions `AEXP_AST` and the boolean expressions `BEXP_AST` since these are fairly standard and of no interest in our presentation.

4.2 Semantic Domains

The next step is to fix the so-called semantic domains, which are the mathematical structures containing the semantical values representing the possible denotations of the syntactic constructs. Figure 4 gives the definitions of these domains as Standard ML datatypes.

Most of these domain equations are quite standard and similar equations can be found in many publications on denotational semantics. They deviate in two aspects from standard definitions. The first is the domain `PARVALUE` for the semantic values of paragraphs. The second is the form of the domain `STATE` modelling the state of a computation. As mentioned above we have to do some nonstandard bookkeeping of continuations. Furthermore, we have seen in the previous section that this bookkeeping has to be done at the end of the paragraphs. Either the continuation of the computation is determined statically by syntactical ordering of the paragraphs, or it is determined dynamically in the case where the flow of control has reached the paragraph through a perform-statement. We model this behavior by assigning a triple of values to a paragraph identifier. The first component of this triple represents the meanings of the statements contained in the paragraph. The second component is a location for continuation with the help of which we will model the dynamically changing computation continuations after paragraphs. The third

```
datatype IDENT_AST        = AstIdent of string
datatype NUMBER_AST       = AstNumber of string
datatype AEXP_AST         = ...
datatype BEXP_AST         = ...
datatype PERFORMCL_AST    = PerformClUntil of BEXP_AST
                          | PerformClTimes of AEXP_AST
datatype PERFORM_AST
         = PerformId     of IDENT_AST
         | PerformIdId   of IDENT_AST * IDENT_AST
         | PerformIdCl   of IDENT_AST * PERFORMCL_AST
         | PerformIdIdCl of IDENT_AST * IDENT_AST * PERFORMCL_AST
datatype STMT_AST
         = StmtSkip
         | StmtMove      of AEXP_AST * IDENT_AST
         | StmtGoto      of IDENT_AST
         | StmtIf        of BEXP_AST * STMTSEQ_AST * STMTSEQ_AST
         | StmtPerform   of PERFORM_AST
and      STMTSEQ_AST  = StmtSeq of STMT_AST * STMTSEQ_AST | StmtSeqEnd
datatype PARGR_AST    = Pargr of IDENT_AST * STMTSEQ_AST
datatype PARGRSEQ_AST = PargrSeq of PARGR_AST * PARGRSEQ_AST | PargrSeqEnd
datatype VARI_AST     = Vari of IDENT_AST * VARI_AST  |  VariEnd
datatype VARISEQ_AST  = VariSeq of VARI_AST * VARISEQ_AST  |  VariSeqEnd
datatype DEC_AST      = Dec of VARISEQ_AST * TYPE_AST
datatype DECSEQ_AST   = DecSeq of DEC_AST * DECSEQ_AST  |  DecSeqEnd
datatype PROGRAM_AST
         = AstProgram of IDENT_AST * DECSEQ_AST * PARGRSEQ_AST
```

Figure 3: Abstract syntax of *MiCo*

component contains the continuation fixed by the syntactical ordering of the paragraphs. This explains the domain equation for PARVALUE. The exact definition of the meaning of a paragraph will be given later by the semantic functions. The domain equation for STATE is motivated by the need to have a means to keep track of the possibly dynamically changing continuations of paragraphs. For this purpose we use the same technique for the locations for continuations as for the locations for storable values. In the same way we have access to the value of a variable through the composition of the environment and the store we also have access to the dynamic continuation of a paragraph identifier, i.e. by looking up

```
datatype IDENTIFIER    = Ident of string
datatype LOC           = Loc of int
datatype CONTLOC       = ContLoc of int
datatype FREELOC       = FreeLoc of int
datatype FREECONTLOC   = FreeContLoc of int
datatype STOREVALUE    = SvEmpty  |  SvInt  of int  |  SvBool of bool
datatype STORE         = Store of (LOC -> STOREVALUE)
datatype STATE         = State of STORE * FREELOC
                                  * CONTSTORE * FREECONTLOC
and      CONTSTORE     = ContStore of CONTLOC -> CONTINUATION
and      ANSWER        = Halt of STATE  |  Error
and      CONTINUATION  = Cont of STATE -> ANSWER
datatype EXPVALUE      = Variable of LOC  |  Constant of STOREVALUE
datatype PARVALUE      = ParCont of (CONTINUATION -> CONTINUATION)
                                    * CONTLOC * CONTINUATION
datatype DENOTVALUE    = Unbound  |  ExpVal of EXPVALUE
                       | ParVal of PARVALUE
datatype ENVIRONMENT   = Env of IDENTIFIER -> DENOTVALUE
datatype ECONTINUATION = Econt of EXPVALUE -> CONTINUATION
datatype DCONTINUATION = Dcont of ENVIRONMENT -> CONTINUATION
```

Figure 4: Semantic Domains

the location of its dynamic continuation in the current environment (second component of the meaning of the paragraph identifier) and then retrieving the actual continuation value in the current state (applying the continuation store to the continuation location).

4.3 Auxiliary Functions and Values

The third step is to provide some auxiliary functions which are helpful in defining the semantic functions. We only need few such functions. Again their definitions and implementations are easy and similar functions (also implemented in Standard ML) can be found [4].

We use polymorphic first and second projection functions `fst : 'a * 'b -> 'a` and `snd : 'a * 'b -> 'b` as well as a normal updating function `subst` of polymorphic type `(''a -> 'b) -> ''a -> 'b -> ''a -> 'b` which returns a new function equal to the function received as a first argument except for one place (second argument to `subst`), where the new function has the third argument of `subst` as result.

We need an allocation function `allocate : STATE -> STATE` which returns a new state with one more free location for storable values. We also use updating functions for locations `update_loc : LOCATION -> CONTINUATION -> ECONTINUATION` and environments `update_env : ENVIRONMENT -> IDENT -> DENOTVALUE -> ENVIRONMENT`. Instead of using left and right values we have a dereferencing function `deref` of type `ECONTINUATION -> ECONTINUATION` which retrieves the storable values in locations if needed. Furthermore, one also needs a normal call-by-value fix-point operator `Y` of type `(('a -> 'b) -> 'a -> 'b) -> 'a -> 'b`.

Clearly, one also has to predefine a set of values such as an initial environment, an end-continuation, and an initial state in order to get a working interpreter.

4.4 Semantic Functions

We now come to the main part of our semantic definition, i.e. the definition of the semantic functions which map syntactical constructs to their semantical values in the above defined domains. The form we present is a continuation style semantics.

For brevity, the presentation is restricted on these parts which deviate from definitions found in standard explanations of denotational semantics. As a meta language for these definitions we use Standard ML. The descriptions should nevertheless be easily understandable even though there are some peculiarities in the source which are mainly due to the strict evaluations strategy of Standard ML. We have also chosen not to give a full semantic error propagation through continuations, but instead use the exception handling mechanism available in Standard ML. We present the definitions in a top-down manner, although Standard ML requires a bottom up approach.

The first semantic functions we show is the one for whole programs in *MiCo*. Its type is:

```
PRG : PROGRAM_AST -> ENVIRONMENT -> CONTINUATION -> CONTINUATION
```

and its definition is (figure 5):

```
fun PRG (AstProgram(AstIdent id, ds, ps)) env cont
    = DS ds env (Dcont(fn env1
        => Cont(fn state0
          => let
               val (env2, state1) = JS ps (env1, state0) cont
               val (Cont cont1)   = PS ps env2 cont
             in
               cont1 state1
             end)));
```

Figure 5: The semantics of *MiCo* programs

The meaning of a program is given by first determining the environment induced by the variable declarations `ds`, which is done in a standard way by the function `DS`. This function has type

`DECSEQ_AST -> ENVIRONMENT -> DCONTINUATION -> CONTINUATION`

and uses itself several other functions:

```
D  : DEC_AST -> ENVIRONMENT -> DCONTINUATION -> CONTINUATION
VS : VARISEQ_AST -> ENVIRONMENT -> DCONTINUATION -> CONTINUATION
V  : VARI_AST -> ENVIRONMENT -> FREELOCATION -> ENVIRONMENT
```

which all have standard definitions.

The next step is to assign the meaning to the identifiers of the paragraphs `ps` by the function `JS` of type

```
PARGRSEQ_AST -> (ENVIRONMENT * STATE) -> CONTINUATION
    -> (ENVIRONMENT * STATE)
```

Recall that we have to assign a triple to each paragraph identifier. The initial values of these triples are computed by `JS`. The exact definition of `JS` is given below.

Finally, the meaning of the paragraph sequence `ps` is determined by the function `PS`.

Let us now explain the function `JS` thereby referring to figure 6. The goal is to initialize the semantical values for paragraph identifiers. Firstly, this means that `JS` has to update the current environment with the meaning of these identifiers. Remember from the subsection on semantic domains that these meanings are triples of type

```
fun JS PargrSeqEnd       es _    = es
  | JS (PargrSeq(p, ps)) es cont
    = let fun F f = fn () =>
        let val cont' = fn state
          =>  let val (Cont ct) = PS ps (fst (f())) cont
              in ct state end
            fun J (Pargr(AstIdent id, ss)) es' cont2 =
              let val State(sto,fl,ContStore csto,FreeContLoc fcl)
                        = snd es'
                  val newstate =
                    State(sto,
                          fl,
                          ContStore(subst csto (ContLoc fcl) cont2),
                          FreeContLoc(fcl+1))
                  val proc = fn retCont => CS ss (fst (f())) retCont
                  val env' = update_env (fst es')
                                        (Ident id)
                                        (ParVal(ParCont(proc,
                                                        (ContLoc fcl),
                                                        cont2)))
             in (env', newstate) end
        in JS ps (J p es (Cont cont')) cont end
      in (Y F) () end
```

Figure 6: The function JS

(CONTINUATION -> CONTINUATION) * CONTLOC * CONTINUATION.

The first component `proc` reflects the meaning of the statement sequence in a paragraph. The second component `(ContLoc fcl)` is the location for the dynamically changing continuations after the execution of a paragraph. The third component `cont2` contains the static continuation induced by the syntactical ordering of the paragraphs.

Secondly, during this initialization process the dynamic continuation is defined equal to the static continuation by updating the continuation part of the old state resulting in a new state.

This explains why JS takes as an argument a (ENVIRONMENT * STATE) pair and returns a new (ENVIRONMENT * STATE) pair. JS uses another function J accomplishing the same task as `JS` but for one paragraph only.

The first case of the function definition of `JS` with an empty sequence of paragraphs is clear by now.

The interesting second case can be explained by the following equations. Let us assume we have a paragraph sequence of the form `PargrSeq(p, ps)` then the following recursive equation is valid:

```
JS (PargrSeq(p, ps)) (env, state) cont =
     JS ps (J p (env, state) cont') cont
```

where `cont'` is equal to

```
PS ps (JS (PargrSeq(p, ps)) (env,state) cont) cont
```

The two points to take into account are

1. to make sure that the meaning of `ps` is evaluated in an environment where all the paragraph identifiers are bound to values, and
2. to make sure to use the right continuation `cont'` after the paragraph `p`.

Now what is the meaning of a paragraph sequence fixed by the function `PS` (figure 7) of type `PARGRSEQ_AST -> ENVIRONMENT -> CONTINUATION -> CONTINUATION` ?

```
fun PS PargrSeqEnd       _    cont = cont
  | PS (PargrSeq(p, _)) env _     = P p env
```

Figure 7: The function `PS`

If we have an empty sequence we simply continue with the rest of the computation `cont`. In the case where we have a paragraph sequence beginning with a paragraph `p` the meaning of the whole sequence is equal to the meaning of the first paragraph `p`. This needs some explanation: Since the program continuation after the paragraph `p` can change dynamically it is not statically determinable which paragraph is executed after `p`. Therefore, as we will see next, the meaning of a paragraph given by the function `P` includes the meaning of its dynamically changing continuation.

The definition of the semantic function `P` for paragraphs is shown in figure 8

```
fun P (Pargr(AstIdent id, ss)) (Env env)
    = CS ss (Env env)
        (Cont(fn (State(sto, fl, ContStore csto, fcl))
          => let val dv = env (Ident id)
             in case dv of (ParVal(ParCont(proc, contloc, cont')))
               => let val (Cont oldcont) = csto contloc
                      val newstate =
                        State(sto,
                              fl,
                              ContStore(subst csto contloc cont'),
                              fcl)
                  in oldcont newstate end
               | _ => raise SEMEX "ParVal expected" end))
```

Figure 8: The function `P`

and has type

`PARGR_AST -> ENVIRONMENT -> CONTINUATION`

The meaning of a paragraph is given by the composition of the meaning of its sequence of statements `ss` and the computation still to be done. The remaining computation is found by retrieving the actual continuation value in the current environment and state. One important point to note is that after reaching the end of a paragraph the value of the dynamic continuation is reset to the statically determined continuation value `cont'` (third component of the meaning of the paragraph identifier). This reflects exactly the behavior found in Cobol programs.

Next we have to describe the semantic functions `CS` of type

`STMTSEQ_AST -> ENVIRONMENT -> CONTINUATION -> CONTINUATION`

for sequences of statements and the semantic function `C` of type

`STMT_AST -> ENVIRONMENT -> CONTINUATION -> CONTINUATION`

of individual statements. We restrict our description to the cases which can not be found in usual denotational semantics. Interesting case are the goto-statement and the different perform-statements.

The listing for the goto-statement is (figure 9):

```
C (StmtGoto(AstIdent id)) (Env env) _
    = Cont(fn (state as State(_, _, (ContStore csto), _)) =>
        let val (Cont cont) =
          let val dv = env (Ident id)
          in case dv of
           (ParVal(ParCont(proc, contloc, _)))
                => proc (csto contloc)
             | _ => raise SEMEX "ParVal expected" end
        in cont state end)
```

Figure 9: The semantics of the goto-statement

A goto-statement simply disregards its own continuation and transfers control to the corresponding paragraph with name `id`. The computation to be done is the composition of the meaning of the sequence of statements `proc` in that paragraph together with the dynamic continuation of the paragraph `(csto contloc)`.

There exists a variety of perform-statements. We start with the one with a single paragraph identifier `id` and no until- or times-clause as shown in figure 10.

```
C (StmtPerform(PerformId(AstIdent id))) (Env env) cont
    = Cont(fn (State(sto, fl, ContStore csto, fcl)) =>
        let val dv = env (Ident id)
        in case dv of
          (ParVal(ParCont(proc, contloc, _)))
            => let val newstate =
                 (State(sto, fl,
                        ContStore(subst csto contloc cont),fcl))
                   val (Cont cont') = proc cont
               in cont' newstate end
        | _ => raise SEMEX "ParVal expected" end)
```

Figure 10: The semantics of `StmtPerform(PerformId(AstIdent id))`

`C` first retrieves the meaning of the paragraph identifier `id`. Then it determines the whole remaining computation by composing the meaning `proc` of the statements in the paragraph `id` with the continuations `cont` of the perform-statement. The continuation cont of the perform-statement is also used in updating the current dynamic continuation value for the paragraph id.

The meaning of the perform-statement with two paragraph identifiers and no additional clause is defined analogously to the one with one paragraph identifier.

Next, we look at the perform-statement with one paragraph identifier id and an until-clause (figure 11).

```
C (StmtPerform(PerformIdCl(AstIdent id,PerformClUntil b))) env cont
    = let val p = StmtPerform(PerformId(AstIdent id))
          fun F g env' cont'
            = let val econt
                = Econt(fn ev => case ev of
                    (Constant (SvBool bc))
                          => if not bc then C p env' (g env' cont')
                                        else cont'
                      | _ => raise SEMEX "Boolean expected"  )
              in B b env' econt end
      in (Y F) env cont end
```

Figure 11: The semantics of StmtPerform(PerformIdCl(.....))

If one is familiar with denotational semantics, it is immediately clear that this statement resembles a while-statement known in other languages, where the statement sequence in the paragraph id is seen as the statements sequence in the while-loop.

Similarly, the perform-statement with one paragraph identifier and a times-clause can be understood as a for-statement.

The semantic values of the perform-statements with two paragraph identifiers and until- or times-clauses can easily be derived from the definitions for perform-statements with one identifier and the corresponding clause.

In order to complete our semantic definition we would also have to give the semantics of arithmetic expressions AEXP_AST and boolean expressions BEXP_AST. In our implementation we used standard definitions which can, for example, be found in [4].

5 Conclusions and Future Work

We illuminated the semantics of control constructs of Cobol by explaining a continuation style denotational semantics for *MiCo*, a language containing the essential control constructs of Cobol. We also provided the main parts of an implementation of this semantics in the functional programming language Standard ML. Thus, this paper contains the central parts for a meta-interpreter for a core of the Cobol 74 programming language. Our analysis clarifies the complex interplay between such constructs as perform-statements, goto-statements and paragraphs or sections.

We are convinced that, based on such formal descriptions of the semantics of the main parts of Cobol, it will be possible to use modern and advanced program analysis techniques in the complex task of re-engineering and reverse engineering. Among these techniques we will further investigate abstract interpretations and continuations passing style program transformations. As one possible application of our semantics we just mention the development of provable semantic invariant program transformations. It is also worth noting that a formal language description of Cobol can be used in future standardization efforts.

To conclude, we remark that in our project we use a slightly extended version of the semantics in order to analyze real world Cobol 74 programs.

References

[1] ANSI, New York. *Programming Language COBOL*, May 1974. ISO 1989-1978.

[2] Micheal J.C. Gordon. *The Denotational Description of Programming Languages.* Springer, 1979.

[3] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall, 1993.

[4] Peter Lee. *Realistic Compiler Generation.* The MIT Press, 1989.

[5] Peter D. Mosses. *Denotational Semantics*, volume B, chapter 11, pages 575 – 632. The MIT Press / Elsevier, 1990. in Vol. B of Handbook of Theoretical Computer Science.

[6] Flemming Nielson and Hanne R. Nielson. *Semantics with Applications.* Wiley, 1992.

[7] Larry C. Paulson. *ML for the Working Programmer.* Cambridge University Press, 1991.

[8] David A. Schmidt. *Denotational Semantics - A Methodology for Language Development.* WM. C. Brown Publishers, 1986.