CHAPTER 5

MORE ON EXCEPTION MECHANISMS

This chapter concerns the specific concept of exception constructs such as goto statements. The material is of narrow interest (the main point is the statement of a number of equivalence theorems) and many readers might choose to omit reading this chapter.

There are two ways of defining exception-like constructs in the denotational style: the Oxford group use so-called 'continuations'; chapter 4 describes the 'exit' approach mostly used in VDM. This difference is perhaps the most substantive issue between the two groups. This chapter defines one language in the most extreme forms of the two styles and proves that the two definitions are equivalent. In fact the proof given here and the discussion in [Bjørner 80b] show that the distinction is not as absolute as might appear at first sight. It should also be mentioned that VDM is not restricted to the exit style.

This chapter is a rewritten version of [Jones 78b] and uses different proof steps. Other relevant papers are [Reynolds 72a] and [Salle 80a].

CONTENTS

5.1 INTRODUCTION

The difficulty of providing a denotational definition of the 'goto' statement is that its effect cuts across the phrase structure of the language. The denotational rule, however, requires that only the denotations of components are to be employed in determining the denotations of compound phrases. The resolution of this difficulty lies in choosing appropriately rich domains as denotations. The Oxford approach is to use functions of higher order than normal transformations. These functions between transformations ("continuations") are explained below. Nothing in the Vienna approach precludes the use of continuations. But, based on earlier work, the Vienna group have preferred to use the *exit* model (explained in chapter 4) where this is adequate. *exit*s are, in fact, weaker than continuations and one of the justifications of this choice is Strachey's own principle of "parsimony in definition tools".

This chapter is built around a small language fragment. The language has been chosen to illustrate the main points at issue while keeping the proof relatively straightforward. Definitions using both *exit*s and continuations are given and a proof that the two definitions are equivalent is outlined. The key to this proof is the intermediate definitions used. These are interesting in their own right since they show that there is not only one difference which distinguishes the *exit* and continuation models. The analysis of these subsidiary decisions shows different possibilities for definitions. Other variations are considered in [Bjørner 80b].

Either definition approach has been shown to be capable of defining the major high-level languages like ALGOL 60 and PL/I. The choice between the approaches must, therefore, be made on other, pragmatic, grounds. Several factors relevant to this choice are pointed out in the course of this chapter.

5.2 THE LANGUAGE

The problems of escape mechanisms in general can be well illustrated by the 'goto' statement. The language used for the comparison is a selection of the essential features of that of chapter 4. What is a 'block' in the full language is called here a 'compound' statement because the declaration of variables is not treated. 'Goto' statements which enter phrase structures are not allowed in the language but the definition tool re-

quired to define this is illustrated in the proof. The proof in [Jones 78b] does include *'cue functions'* in the language definition. Labels are assumed to be unique throughout the program. The abstract syntax is:

Program :: *Stmt*
Stmt = *Compound* | *Goto* | *Assign*
Compound :: *Namedstmt**
Namedstmt :: [*Id*] *s-body*:*Stmt*
Goto :: *Id*
Assign :: *Varref Expr*

The context conditions are as in chapter 4 (mutatis mutandis) with the additional constraint that labels are unique throughout the program. Use is made of:

dlabs: Namedstmt* → *Id-set*

A new function to collect all (rather than just the direct) labels is defined:

labs: Stmt → *Id-set*
labs[*mk-Goto*(*id*)] ≜ {*id*}
labs[*mk-Assign*(*vf*,*e*)] ≜ {}
labs[*mk-Compound*(*nsl*)] ≜
  contndll[*nsl*] ∪ $\underline{union}${*labs*[*s-body*(*nsl*[*i*])] | *i* ∈ $\underline{inds}$ *nsl*}

The function from chapter 4 which selects a portion of a list based on an identifier is extended to cover labels which are not direct. The result is a sub-list of the argument where the given identifier is in the (indirect) labels of the head of the list:

sel: Id × *Namedstmt** → *Namedstmt**
pre-sel[*id*,*nsl*] = *id* ∈ *labs*[*nsl*]

The semantic definition given in this section follows that in chapter 4. The domain *STATE* can be taken as given and its detailed structure can be hidden behind a function:

assign: Varref × *Expr* → *STATE* ≅ *STATE*

The advantage of insisting that labels be globally unique is that no

Activation identifiers are required in the definition. Thus necessary transformations (called *Xtr* to link them to the *exit* definition) are:

$$Xtr = STATE \rightarrow STATE \times [Id]$$

The semantic functions are derived from those in chapter 4 in an obvious way. Once again, *X* is used to identify the definition approach. The definition given here uses the *exit* combinators. These are expanded and the types of all functions shown explicitly in the next-but-one section.

$$X[mk\text{-}Program(s)] \triangleq X[s]$$

$$X[mk\text{-}Compound(nsl)] \triangleq tixe\ [id \mapsto Xl[sel[id,nsl]] \mid id \in dlabs[nsl]]\ in\ Xl[nsl]$$

$$Xl[\langle\rangle] \triangleq I_{STATE}$$

$$Xl[nsl] \triangleq X[s\text{-}body(hd\,nsl)];Xl[tl\,nsl]$$

$$X[mk\text{-}Goto(id)] \triangleq exit(id)$$

$$X[mk\text{-}Assign(vr,e)] \triangleq assign(vr,e)$$

It is a property of this definition that the only "goto" *exit*s which are not resolved by *X* are to non-local labels. Thus:

$$(X[s](\sigma) = (\sigma',a)) \supset ((a=nil) \vee \neg(a \in labs[s]))$$

It follows from the context condition, which requires that all labels are defined, that:

$$(X[mk\text{-}Program(s)](\sigma) = (\sigma',a)) \supset (a=nil) \vee \neg(a \in labs[s])$$

The fact that all labels are handled in this way could be formalized by writing:

$$X[mk\text{-}Program(s)] \triangleq (\lambda\sigma,a.\sigma) \circ X[s]$$

reducing the *Xtr* to an object in:

$$STATE \rightarrow STATE$$

One of the advantages of an *exit*-style definition is the way in which the effect of exits is automatically localized. It is clear that any compound, with only local branches, has one of the simpler transformations as its denotation.

A further effect of the use of *exit* combinators is that the impact of the 'goto' statement on the definition is limited. An early 'functional semantics' for ALGOL 60 ([Allen 72a]) shows how the use of the *exit* concept can permeate a complete definition without the use of combinators.

5.3 CONTINUATIONS

The alternative and more widely used approach to the definition of exception constructs is to use 'continuations'. These were invented independently by F.L.Morris [Morris 70a] and C.Wadsworth [Strachey 74a]. As with the *exit* approach, continuations resolve the problem of conforming to the denotational rule by complicating the domain of the denotations. Technically the idea is to move from simple transformations:

$$TR = STATE \xrightarrow{\sim} STATE$$

to the higher order:

$$TR \rightarrow TR$$

In order to clarify which definition method is being used at any time, this chapter uses:

$$CONT = STATE \xrightarrow{\sim} ANS$$
$$CTR = CONT \rightarrow CONT$$

The use of the answer domain (*ANS*) is explained in the proof - the reader can equate it to *STATE* in this section.

The idea behind these higher-order functions is that the "total transformation" is defined in the context of a "remaining transformation". Thus:

$$C: Stmt \rightarrow CTR$$
$$C[\![s]\!](\theta) \triangleq \theta'$$

(Parameters of continuation type are traditionally enclosed in braces rather than parentheses - this aid to readability is preserved here.) Both θ and θ' are members of:

$$STATE \xrightarrow{\sim} STATE$$

The meaning of s, where θ is to be done next, is θ'. 'Goto' statements will actually be given a denotation which shows the passed continuation being ignored; they do, however, require denotations for labels to be stored in an environment:

$$ENV = Id \xrightarrow[m]{} CONT$$

Thus the actual semantic functions (with C for continuation) become:

$$C: Stmt \to ENV \to CTR$$

$$C[mk\text{-}Program(s)] \triangleq$$
$$\underline{let}\ \rho = [id \mapsto Crest[id,s](\rho)\{I_{STATE}\} \mid id \in labs(s)]\ \underline{in}$$
$$C[s](\rho)\{I_{STATE}\}$$

Which is of the type:

$$Program \to STATE \xrightarrow{\sim} STATE$$

Then:

$$C[mk\text{-}Compound(nsl)] \triangleq Cl[nsl]$$
$$Cl[\langle\rangle](\rho)\{\theta\} \triangleq \theta$$
$$Cl[nsl](\rho)\{\theta\} \triangleq C[s\text{-}body(\underline{hd}\,nsl)](\rho)\{Cl[\underline{tl}\,nsl](\rho)\{\theta\}\}$$

which is of type:

$$Named\text{-}stmt^* \to ENV \to CTR$$

$$C[mk\text{-}Goto(id)](\rho)\{\theta\} \triangleq \rho(id)$$
$$C[mk\text{-}Assign(vr,e)](\rho)\{\theta\} \triangleq \theta \circ assign(vr,e)$$

The function which is used to derive the continuation for the "rest" of the execution from a particular label is only used for compound statements:

$$Crest[id, mk\text{-}Compound(nsl)](\rho)\{\theta\} \triangleq$$
$$\quad \underline{if}\ id \in dlabs(nsl)\ \underline{then}\ Cl[sel(id,nsl)](\rho)\{\theta\}$$
$$\quad \underline{else}\ (\underline{let}\ rl = sel(id,nsl)\ \underline{in}$$
$$\qquad Crest[id, s\text{-}body(\underline{hd}\, rl)](\rho)\{Cl[\underline{tl}\, rl](\rho)\{\theta\}\})$$

Which is of type:

$$Id \times Compound \to ENV \to CTR$$

The reader who finds this whole definition back to front is to be sympathized with. The proponents of the continuation approach will offer reassurance that this way of thinking eventually becomes natural.

Once the initial difficulties of adjustment are overcome, there remain some concerns about continuation definitions.

The label denotations stored in the environment all represent the effect of starting execution at that label and continuing to the end of the whole program. The result of this is that the denotation of a block with no non-local '*goto*' statements is still of type:

$$ENV \to CTR$$

Since the environment is computed at the program level, the denotation is not closed in the way it is with *exits* (*nil* label returned). What appears to be lost in the definition is the fact that the continuation used to develop the denotations for the labels in a block is the same as the continuation for statements in the block.

There is another potential problem which does not manifest itself in this language.

The need for epilogue type action is discussed in chapter 4; it is clear how such transformations are composed with the denotation of a block using the *exit* approach; such transformations do not fit naturally with the continuation approach. The difficulty is that the denotation of a label (in so far as the number of blocks to be closed is concerned) has to vary from one block to another. Although somewhat messy, it is in fact possible to compose such actions onto label denotations even in a continuation definition (see [Bjørner 80b]).

5.4 PROOF PLAN

The preceding two sections each offer definitions which the remainder of this chapter shows to be 'equivalent'. Formally the result is that the *X* and the *M* semantics give the same transformation (as the denotation) for any (valid) program.

Careful analysis of the two definitions shows that the overall dissimilarity can be divided into three areas:

(i) In the continuation definition, the denotation which is associated with a label via the environment reflects the effect of starting execution at that label and continuing to the end of the entire program. The *exit* definition, however, provides denotations for labels which reflect only the transformation corresponding to the execution from the label to the end of its corresponding compound statement.

(ii) The mode of generating the denotations in the two approaches differs: continuations are built up from a remaining continuation composing "backwards"; in the exit definition the composition is "forwards" from the label.

(iii) The continuation definition passes the denotations of labels in the environment whereas the meaning of labels is used in the *exit* definition (by the *tixe* combinator) at the level of the compound statement.

The proof method to be used is to make these changes singly and to show that each of the definitions is equivalent in the required sense. The overall result obviously follows by transitivity. The first contribution to making the semantics appear more similar is to expand the combinator definitions used in the *exit* version. Thus what follows is simply a rewriting of the exit definition expanding the combinators of chapter 4.

$$X[mk\text{-}Program(s)] \triangleq X[s]$$

As indicated above, this can be shown to give a result of type:

$$STATE \to STATE \times \{nil\}$$

The main semantic functions are of type:

$X: Stmt \to Xtr$

$$X[mk\text{-}Compound(nsl)] \triangleq$$
$$\underline{let}\ \rho = [id \mapsto Xl[sel(id,nsl)] \mid id \in dlabs(nsl)] \quad \underline{in}$$
$$\underline{let}\ r = \lambda\sigma,a.\underline{if}\ a \in \underline{dom}\rho\ \underline{then}\ r(\rho(a)(\sigma))\ \underline{else}\ (\sigma,a)\ \underline{in}$$
$$r \circ Xl[nsl]$$

$$Xl[<>] \triangleq \lambda\sigma.(\sigma,\ \underline{nil})$$

$$Xl[nsl] \triangleq (\lambda\sigma,a.\underline{if}\ a=\underline{nil}\ \underline{then}\ Xl[\underline{tl}nsl](\sigma)\ \underline{else}\ (\sigma,a)) \circ X[s\text{-}body(\underline{hd}nsl)]$$

$$X[mk\text{-}Goto(id)] \triangleq \lambda\sigma.(\sigma,id)$$

$$X[mk\text{-}Assign(vr,e)] \triangleq \lambda\sigma.(assign(vr,e)(\sigma),\underline{nil})$$

5.5 EXITS HANDLED GLOBALLY

One of the arguments used in favour of the *exit* style definition is the way in which the effect of 'goto' statements can be localized to the compound statement in which the label occurs. It is, however, possible to write a definition in which the handling of *exit*s is done globally. This is a first step towards the overall equivalence and also provides an opportunity to illustrate '*cue*-functions'. The name *cue* has been used because these functions are prompted to begin execution at a particular label. Here they are used to create a transition (of Xtr) from the label to the end of the whole program. In general, *cue*-functions can be used in defining languages which permit 'goto' statements to enter phrase structures.

The E semantic functions given here are of the same types as the corresponding X functions.

$$E[mk\text{-}Program(s)] \triangleq$$
$$\underline{let}\ \rho = [id \mapsto Ecue[id,s] \mid id \in labs(s)] \quad \underline{in}$$
$$\underline{let}\ r = \lambda\sigma,a.\underline{if}\ a \in \underline{dom}\rho\ \underline{then}\ r(\rho(a)(\sigma))\ \underline{else}\ (\sigma,a)\ \underline{in}$$
$$r \circ (s)$$

$$E[mk\text{-}Compound(nsl)] \triangleq El[nsl]$$

$$El[<>] \triangleq \lambda\sigma.(\sigma, \underline{nil})$$

$$El[nsl] \triangleq (\lambda\sigma,a.\underline{if}\ a=\underline{nil}\ \underline{then}\ El[\underline{tl}nsl](\sigma)\ \underline{else}\ (\sigma,a))\circ E[s\text{-}body(\underline{hd}nsl)]$$

$$E[mk\text{-}Goto(id)] \triangleq \lambda\sigma.(\sigma, id)$$

$$E[mk\text{-}Assign(vr,e)] \triangleq \lambda\sigma.(assign(vr,e)(\sigma), \underline{nil})$$

$$Ecue[id,mk\text{-}Compund(nsl)] \triangleq$$
$$\underline{if}\ id \in labs(nsl)\ \underline{then}\ El[sel(id,nsl)]$$
$$\underline{else}\ (\underline{let}\ rl = sel(id,nsl)\ \underline{in}$$
$$(\lambda\sigma,a.(\underline{if}\ a=\underline{nil}\ \underline{then}\ El[\underline{tl}rl](\sigma)\ \underline{else}\ (\sigma,a))\circ$$
$$Ecue[id,s\text{-}body(\underline{hd}rl)]$$

This definition can be shown to be equivalent to the *X* semantics.

The first step is to relate the two definitions of *Compound* statement semantics by a lemma. The intention is to show that the *X* semantics are the same as the *E* semantics providing an extra *exit* trap is placed around the latter. This relies on a hypothesis about the ρ argument used in the trap:

$$(\forall id \in labs(cpd))$$
$$(\rho(id) = Ecue[id,cpd]) \supset$$
$$(\underline{let}\ r = \lambda\sigma,a.(\underline{if}\ a \in \underline{dom}\rho\ \underline{then}\ r(\rho(a)(\sigma))\ \underline{else}\ (\sigma,a))\ \underline{in}$$
$$r\circ E[cpd])$$
$$= X[cpd]$$

This lemma can be proved by induction on the depth of nesting of *Compound* statements. It is an immediate corollary of the lemma that for (valid) *Programs*:

$$E[mk\text{-}Program(s)] \triangleq X[mk\text{-}Program(s)]$$

This follows since the hypothesis of the lemma is discharged by the definition of ρ in *E*.

5.6 CONTINUATIONS WITHOUT AN ENVIRONMENT

The next step brings in a continuation-like treatment of 'goto' statements. That is, the second difference listed in the section on "Proof Plan" is resolved in the direction of building up label denotations by composing from the back. In order to achieve this an extra (continuation) argument has to be passed to the *D* semantic functions. In this definition, however, label denotations are not passed in an environment. Furthermore, the objects which are passed and returned are not "normal" continuations: the freedom of defining the result of *CONT* to be *Ans* is used here to give:

$$D: Stmt \to XTR \to XTR$$

Thus:

$$D[mk\text{-}Program(s)] \triangleq$$
$$\quad \underline{let}\ r = \lambda\sigma,a.\underline{if}\ a \in labs(s)\ \underline{then}\ r(Dcue[id,s])\{\lambda\sigma.(\sigma,\underline{nil})\}(\sigma)$$
$$\qquad\qquad \underline{else}\ (\sigma,a)\ \underline{in}$$
$$\quad r \circ D(s)\{\lambda\sigma.(\sigma,\underline{nil})\}$$

$$D[mk\text{-}Compound(nsl)]\{\theta\} \triangleq Dl(nsl)\{\theta\}$$

$$Dl[<>]\{\theta\} \triangleq \theta$$

$$Dl[nsl]\{\theta\} \triangleq D[s\text{-}body(\underline{hd}\,nsl)]\{Dl[\underline{tl}\,nsl]\{\theta\}\}$$

$$D[mk\text{-}Goto(id)]\{\theta\} \triangleq \lambda\sigma.(\sigma,id)$$

$$D[mk\text{-}Assign(vr,e)]\{\theta\} \triangleq \lambda\sigma.\theta(assign(vr,e)(\sigma))$$

$$Dcue[id,mk\text{-}Compound(nsl)]\{\theta\} \triangleq$$
$$\quad \underline{if}\ id \in dlabs(nsl)\ \underline{then}\ Dl[sel(id,nsl)]\{\theta\}$$
$$\quad \underline{else}\ (\underline{let}\ rl = sel(id,nsl)\ \underline{in}$$
$$\qquad Dcue[id,s\text{-}body(\underline{hd}\,rl)]\{Dl[\underline{tl}\,rl]\{\theta\}\})$$

This definition can now be shown to be equivalent to the *E* semantics of the last section. As with the proof in that section, it is convenient to separate a lemma. This lemma states that the *Dl* semantics with an (continuation) argument θ is equivalent to *El* with θ used after it in the case of normal *exit*:

$$(\lambda\sigma,a.\ \underline{if}\ a=\underline{nil}\ \underline{then}\ \theta(\sigma)\ \underline{else}\ (\sigma,a))\circ El[nsl] = Dl[nsl]\{\theta\}$$

This lemma is again proved by induction on the depth of compound statements. But, since a subsidiary induction is required, the basis is sketched here. For the basis, it is assumed that no elements of *nsl* are compound statements. It is then possible to use induction on the length of *nsl*. If *nsl* is empty:

$$\begin{aligned}(\lambda\sigma,a.\underline{if}\ a=\underline{nil}\ \underline{then}\ \theta(\sigma)\ \underline{else}\ (\sigma,a))\circ El[<>] &= \lambda\sigma.(\theta(\sigma))\\ &= \theta\\ &= Dl[<>]\{\theta\}\end{aligned}$$

If *nsl* is not the empty list a case distinction is required. Assume:

$$s\text{-}body(\underline{hd}nsl) \in Goto$$

$$E[mk\text{-}Goto(id)] = \lambda\sigma.(\sigma,id)$$

$$\begin{aligned}El[nsl] &= (\lambda\sigma,a.\underline{if}\ a=\underline{nil}\ \underline{then}\ El[\underline{tl}nsl](\sigma)\ \underline{else}\ (\sigma,a))\circ E[mk\text{-}Goto(id)]\\ &= \lambda\sigma.(\sigma,id)\end{aligned}$$

$$(\lambda\sigma,a.\underline{if}\ a=\underline{nil}\ \underline{then}\ \theta(\sigma)\ \underline{else}\ (\sigma,a))\circ El[nsl] = \lambda\sigma.(\sigma,id)$$

$$\begin{aligned}Dl[nsl]\{\theta\} &= D[mk\text{-}Goto(id)]\{Dl[\underline{tl}nsl]\{\theta\}\}\\ &= \lambda\sigma.(\sigma,id)\end{aligned}$$

Now, the only other possibility is that:

$$s\text{-}body(\underline{hd}nsl) \in Assign$$

$$E[mk\text{-}Assign(vr,e)] = \lambda\sigma.(assign(vr,e)(\sigma),\underline{nil})$$

$$\begin{aligned}El[nsl] &= (\lambda\sigma,a.\underline{if}\ a=\underline{nil}\ \underline{then}\ El[\underline{tl}nsl](\sigma)\ \underline{else}\ (\sigma,a))\\ &\quad \circ E[mk\text{-}Assign(vr,e)]\\ &= (\lambda\sigma.El[\underline{tl}nsl](assign(vr,e)(\sigma)))\end{aligned}$$

$$\begin{aligned}Dl[nsl]\{\theta\} &= D[mk\text{-}Assign(vr,e)]\{Dl[\underline{tl}nsl]\{\theta\}\}\\ &= \lambda\sigma.\{Dl[\underline{tl}nsl]\{\theta\}\}(assign(vr,e)(\sigma))\end{aligned}$$

which, by induction hypothesis:

$$= \lambda\sigma.(\lambda\sigma,a.\underline{if}\ a=\underline{nil}\ \underline{then}\ \Theta(\sigma)\ \underline{else}\ (\sigma,a))^{\circ}$$
$$El[\underline{tl}\,nsl](assign(vr,e)(\sigma))$$

$$= (\lambda\sigma,a.\underline{if}\ a=\underline{nil}\ \underline{then}\ \Theta(\sigma)\ \underline{else}\ (\sigma,a))^{\circ}El[nsl]$$

Here again, the main result is a corollary of the lemma; for programs which are *Compound* statements:

$$D[mk\text{-}Program(mk\text{-}Compound(nsl))]$$
$$= \underline{let}\ r = \ldots\ \underline{in}\ r^{\circ}Dl[nsl]\{\lambda\sigma.(\sigma,\underline{nil})\}$$
$$= \underline{let}\ r = \ldots\ \underline{in}\ r^{\circ}(\lambda(\sigma,a).(\sigma,\underline{nil}))^{\circ}El[nsl]$$
$$= \underline{let}\ r = \ldots\ \underline{in}\ r^{\circ}El[nsl]$$
$$= E[mk\text{-}Program(mk\text{-}Compound(nsl))]$$

5.7 NORMAL CONTINUATIONS

As the reader may have noticed, the names of the semantic functions have been progressing back through the alphabet. The remaining task is to establish the connection between the *D* semantics of the preceding section and the original continuation semantics (*C*).

Here again, it is convenient to separate a lemma which relates *Dl* and *Cl*. With:

$$cons = \lambda id.\lambda\sigma.(\sigma,id)$$

the statement of the lemma is:

$$r^{\circ}Dl[nsl]\{\Theta\} = Cl[nsl](r^{\circ}cons)\{r^{\circ}\Theta\}$$

which shows that in order to perform *r* after the (*Dl*) meaning of *nsl*, it is necessary to compose *r* both with the passed continuation and with the environment entries. The proof follows the same pattern as that of the lemma in the preceding section. The final result can then be shown as follows:

$$D[mk\text{-}Program(mk\text{-}Compound(nsl))]$$
$$= \underline{let}\ r = \ldots\ \underline{in}\ r^{\circ}Dl[nsl]\{\lambda\sigma.(\sigma,\underline{nil})\}$$
$$= \underline{let}\ r = \ldots\ \underline{in}\ Cl[nsl](r^{\circ}cons)\{r^{\circ}\lambda\sigma.(\sigma,\underline{nil})\}$$
$$= \underline{let}\ \rho = [id \mapsto Crest[id,nsl](\rho)\{\lambda\sigma.(\sigma,\underline{nil})\} \mid id \in labs(nsl)]\ \underline{in}$$
$$Cl[nsl](\rho)\ \{\lambda\sigma.(\sigma,\underline{nil})\}$$

Since it has been shown above that the transformation always yields a *nil* second component, it is only necessary to make a systematic change to reduce the *ANS* domain to *STATE* and get:

$$C[mk\text{-}Program(mk\text{-}Compound(nsl))]$$

This concludes the chain of equivalence proofs:

$$X = E = D = C$$

A few remaining comments can be made about the comparison of the two definitions, *X* and *C*. The limitation of the current proof should be clearly understood. Two language definitions have been shown to be equivalent. This does not show that the two approaches (i.e. *exit*s and continuations) are equivalent. In fact, it would appear that definitions using *exit*s are less powerful in that they cannot (readily) be used to define features like coroutines where the essence is to pass a a continuation. There would appear to be an argument that a language definition is made more perspicuous if the domains exactly match the power of the language and this argument has been used to justify the choice of *exit*s to define "goto"-like exceptions. It should at least cause some hesitation to those who insist that "standard" semantics implies the use of continuations. An advantage of definition by continuations is that the *Answer* domain can be used to yield some final result other than (or as well as) the state. For example, [Gordon 79a] delivers the output file in this way. The definition of ALGOL 60 in Chapter 6 somewhat clouds the possible operations on output files. This deficiency can readily be overcome by a simple redefinition of the combinators.

One of the purposes of a formal definition is to provide a criterion for the correctness of implementations. Proofs have been based on both styles of definition but no detailed comparison has yet been published.