DENOTATIONAL SEMANTICS OF GOTO:

AN EXIT FORMULATION AND ITS RELATION TO CONTINUATIONS

Cliff B.Jones

Abstract:

This paper discusses the problem of providing a definition for the "GOTO" statement within the framework of denotational semantics. The accepted approach to the problem is to use "Continuations". An alternative "Exit Formulation" is described in this paper. A small language is introduced which illustrates the difficulties caused by statements which terminate abnormally. For this language definitions based on both approaches are provided. A proof of equivalence of the two definitions is then given. In a closing discussion it is pointed out that continuations can define a wider class of languages than exits, although the latter have been shown to be adequate to define languages as complex as PL/I.

CONTENTS

Figures:

1. INTRODUCTION

There exists by now a considerable body of work on the formal definition of programming languages (see Lucas 78, in this volume, for a historical review). Most of the work can be categorised as either "abstract interpreters", "denotational semantics" or "axiomatic". This paper attempts to make a contribution to the understanding of denotational semantics. This approach is particularly associated with the Programming Research Group at Oxford University. Evolving from the earlier work on abstract interpreters (see Lucas 69) the more recent work of the Vienna Laboratory has also used the denotational style (see Bekić 74).

A language can be given by the set of its texts. To define the semantics of a language one must associate a meaning or denotation with each text in the set. Since the set of texts will, for interesting languages, be of infinite cardinality, this link will be shown by defining a function from the set of texts to a set of denotations. For such a definition of a large language to be comprehensible, it is required that the denotation of a compound text should depend solely on the denotations of its component parts. (Clearly, this rule must be applied only to a sensible level: ascribing meaning to single characters and then trying to construct the meaning of identifiers and keywords is unlikely to prove illuminating. The limit of sensible decomposition is usually indicated by the abstract syntax of a language.) In order for a function from texts to denotations to define the semantics of the texts, it is obviously a pre-requisite that the denotations themselves should be objects with known meanings. One characteristic of denotational semantics is the use of mathematical objects (especially functions) as the denotations. This accounts for the alternative name of "mathematical semantics". In fact the functions chosen as denotations are of a very general form and it has been a considerable task to show that such functions do indeed have a consistent meaning (see Scott 71).

There is, within the "denotational school", agreement as to the concepts required to provide definitions of simple languages. (However, a number of notational differences lead to differences in the appearance of conceptually similar definitions, see next section). Remembering the "denotational rule" that denotations of composite objects should be built from the denotations of their components, the following observations can be made. For a purely functional language it is easy to agree a definition; for a language which includes an assignment construct the concept of a store (i.e. a mapping from identifiers to values) permits denotations to be defined which are functions from stores to stores; the generally accepted approach to block structured languages is to introduce locations and environments. Whilst refs Mosses 74 and Bekić 74 build from this basic list of agreements and tackle similar large languages, an important difference can be found.

The important difference between the Oxford and Vienna groups can be found in their approach to problems of abnormal termination. The archetypal problem in this category is the "goto" statement. A definition conforming to the denotational rule is difficult to construct for languages which include goto statements precisely because their effect is to transfer control across the structure over which the denotations are being constructed. (This power of goto statements has led to a movement for their elimination. This controversy is not entered into here. Rather, models are explained which are general enough to model goto. On such models one can then compare alternative language constructs which might offer the desirable features, without the danger, of goto statements. Of course, a language feature may eventually be selected for which simpler models are possible. What the models here offer is a basis from which to work.) The Oxford group use "continuations" (see Strachey 74) to define goto-like constructs: the definition in section 5 below is in this style. The same small example language is defined using the Vienna "exit" approach in section 4. The language itself is introduced informally in section 3 after some comments on notation. Given two definitions of the same language, the question of their relationship can be posed: equivalence is proved in section 6. Whilst both continuations and exits have been shown to be powerful enough to define commonly used programming languages, the two approaches are not of equivalent power, section 7 contains some concluding comments on this point.

2. NOTATION

The basis of the notation to be used in this paper is taken from logic and lambda calculus and will probably be familiar from the literature. The items of special interest within this paper are introduced below as required. A more complete explanation is available elsewhere in this volume (Jones 78a); Jones 75 provides a stepwise development of the exit concept.

One of the most dangerous traps when comparing two languages is to let the superficial syntactic differences confuse the real issue which is that of meaning. The difference in appearance between the Oxford and Vienna definitions is very striking. The former group has achieved succinctness in order to facilitate formal reasoning about smaller definitions, whilst the tasks tackled by the Vienna group have led them to strive for readability. This paper, being based on a very small language, compromises a little for the sake of compactness. Thus short names are used for the syntactic classes and "<>" is used instead of a named (tree) constructor where context makes the choice clear (the convention for dropping semantic rules for syntax classes defined to be a list of alternatives is also followed). Other than this the definitions, even that by continuations, are given in a Vienna-style. The issue to be reviewed is the differences between the domains and function types. Choices like the use of different bracket symbols, explicit versus implicit typing of functions and the degree of abstractness for the syntax might influence the number of characters in a definition but would, if used on one of the definitions, serve only to cloud the main distinction.

3. A SMALL LANGUAGE

This section introduces the language which will be used as the basis for the remainder of the paper. The basic statements of the language are "goto" and an unanalyzed class of elementary statements. About these latter all necessary knowledge is given by the function "el-sem" which associates a state-transformation (i.e. a function over the class Σ) with each member of El-stmt. Statements can be optionally named and lists thereof can be formed into compound statements. Such compound statements are also statements and thus can be named and used as elements of other lists. The abstract syntax of the language is given, along with a full name for each class of objects to aid comprehension, in *fig. 1*. The structure of the classes Id and El-stmt is not further defined.

Note: In this paper a superscript ^{0} is to be read as the functional composition operator. It is elsewhere represented by the fat dot: •.

Full name	Abstract syntax
Program	P :: C
Compound statement	C :: s-b:Ns*
Named statement	Ns :: s-n:[Id] s-b:S
Statement	S = C | G | E
Goto statement	G :: Id
Elementary statement	El :: El-stmt
Identifiers	Id

Fig. 1: The Language to be Defined

In order to facilitate discussion of the identifier prefixes of statements, two predicates which check for the (direct and indirect) containment of identifiers and a function yielding the index of that statement which contains an identifier (under the assumption that it is contained somewhere) are introduced:

$is\text{-}dcont(id,nsl) \Leftrightarrow (\exists i \in \{1:\underline{len}nsl\})(s\text{-}n^{0}nsl(i) = id)$

$type:\ Id\ Ns^{*} \rightarrow Bool$

$is\text{-}cont(id,nsl) \Leftrightarrow (is\text{-}dcont(id,nsl) \vee$
$\qquad (\exists i \in \{1:\underline{len}nsl\})(s\text{-}b^{0}\text{-}nsl(i) \in C \ \&\ is\text{-}cont(id,s\text{-}b^{0}s\text{-}b^{0}nsl(i))))$

$type:\ Id\ Ns^{*} \rightarrow Bool$

$ind(id,nsl)=$
$\quad is\text{-}dcont(id,nsl) \rightarrow (\iota i)(s\text{-}n^{0}nsl(i)=id)$
$\quad T \rightarrow (\iota i)(s\text{-}b^{0}nsl(i) \in C \ \&\ is\text{-}cont(id,s\text{-}b^{0}s\text{-}b^{0}nsl(i)))$

$type:\ Id\ Ns^{*} \rightarrow Nat$

$pre:\ is\text{-}cont(id,nsl)$

It is assumed that well-formed programs satisfy the context condition that all label identifiers used in goto statements are contained exactly once within the program. With respect to the statement list within which the goto is placed, the target statement may be within the same list, within a containing list or within a compound statement which is a member of the same list, The local "hop" and the abnormal exit from a list should require no comment. The ability to jump into a phrase structure is allowed, because it is included in many programming languages (cf. Algol 60 "goto" into branches of conditional statements.)

The choice of features in this language has been made with some care in order to exhibit most of the complexity of large languages in a framework of reasonable size. Thus the ability to hop between elements of a list has been supplemented by permitting goto statements to enter and leave syntactic units. In fact if the reader compares this language to Algol 60 (see Henhapl 78 in this volume) only the ability to pass labels and procedures as parameters forces an extension of the ideas used here. Abnormal termination of a block via a goto statement is a straightforward extension and the problems associated with redefinition of names in a block-structured language can be solved in a uniform way for variable and label identifiers (see Bekić 74 for a discussion of label variables).

The elementary statements of the language are assumed to cause changes to a class of states (Σ):

el-sem: El-stmt $\rightarrow$ ($\Sigma \rightarrow \Sigma$)

Were it not for the inclusion of the "goto" construct, it would be straightforward to provide a definition which associated a transformation with any elements of S. (The denotation of a list of statements being the composition of the denotations of the elements of the list.) Whilst as a result of the context condition given above, the denotation of a whole program will be such a transformation, it is not possible to ascribe such a simple denotation to "goto" statements. The next two sections offer different solutions to this problem.

4. DEFINITION BY EXIT

The difficulty of finding a suitable definition for a language which includes goto statements is that its ability to cut across syntactic units forces changes on the semantics of all such units. A motivation of the exit approach to be defined in this section was to minimize the effect of these changes on the overall appearance of a definition. The key to achieving the desired effect without writing it into all of the semantic equations is to define appropriate combinators. Thus in a simple language (i.e. one without goto statements) a combinator denoting functional composition might be written ";". If this same symbol is reinterpreted as the more complex combinator used below, a definition for a language with goto statements can preserve a simpler appearance except, of course, for those semantic equations which deal specifically with goto statements.

The basic idea of definition by exit is to associate a denotation with each statement which is a function of type:

$$E \;=\; \Sigma \rightarrow \Sigma\, A$$

where:

$$A \;=\; Id \mid \underline{NIL}$$

Thus the denotation of a statement is a function from states to pairs: the first element is a state and the second is either an identifier or $\underline{NIL}$. In the case that applying the denotation of a statement to a state results in no "goto", the respective range element will be the result state paired with $\underline{NIL}$. If, however, a "goto" is encountered to a label not contained within the statement, the range element will pair the state reflecting state transition up to the time of the "goto" with the target label.

The definition, using combinators whose meaning is made precise below, is given in *fig 2*. The function names all begin with "*x*-" to signify that they are part of the *exit*-style definition. It is not difficult to provide an intuitive understanding of this definition. The function *x-g* which defines the semantics of goto statements uses the "*exit*" combinator which simply pairs the argument state with the given (identifier) value. Wherever a simple $(\Sigma\rightarrow\Sigma)$ function is shown it is interpreted as yielding $\underline{NIL}$ paired with whatever the output state

```
x-p(<cp>) = x-c(cp)


x-c(cp) = x-cp(NIL,cp)


x-cp(ido,<nsl>) =
    tixe [ id→x-l(id,nsl) | is-cont(id,nsl) ]
    in x-l(ido,nsl)


x-l(ido,nsl) =
    ido=NIL              → x-nsl(nsl,1)
    is-dcont(ido,nsl)→ x-nsl(nsl,ind(ido,nsl))
    T                    → (let i = ind(ido,nsl)
                            x-cp(ido,s-b(nsl(i)));
                            x-nsl(nsl,i+1)
                           )


x-nsl(nsl,i) =
    i≤lennsl  → (x-s(s-b(nsl(i)));x-nsl(nsl,i+1))
    T         → I


x-g(<id>) = exit(id)


x-el(<el>) = el-sem(el)
```

fig 2: Definition using exit combinators

would have been. This explains *x-el* and the second case of *x-nsl* (*I* is the identity on Σ). The fact that these denotations, and thus that of the excised *x-s*, are of type *E* force their combination with one another to be more complex. The ";" combinator applies the second *E* transformation only if the second component of the first result is *NIL*, otherwise the result of the two composed transformations is exactly that of the first. It remains only to explain *x-cp*. Here the combinator "*tixe*" (spell back-to-front!) is for the converse situation from ";". If a normal pair (i.e. *NIL* second component) is the result of the *in* transformation nothing more is done; the first mapping defines, for some restricted set of exit values, the action to be taken if the *in* transformation returns a non-*NIL* result. It is important to realize that this mapping covers a finite number of cases which can be determined from the text being defined.

The types of the semantic functions can be given:

x-p: $P \to E$
x-c: $C \to E$
x-cp: $[Id]\ C \to E$
x-l: $[Id]\ Ns^* \to E$
x-nsl: $Ns^*\ Nat \to E$
x-s: $S \to E$ (assumed)
x-g: $G \to E$
x-el: $El \to E$

The formal meanings of the combinators is now given. The format used for these definitions is first to list any assumptions, then show the type of the combinator expression (after a ":") and finally to provide the definition (after "$\triangleq$").

Firstly the *exit* combinator:

for $id \in Id$
$\underline{exit}(id) : E$
$\underline{exit}(id) \triangleq \lambda\sigma.\langle\sigma, id\rangle$

The promotion of a simple transformation to one of type *E* is governed by context:

for $t : \Sigma \to \Sigma$ in a context requiring E
$t : E$
$t \triangleq \lambda\sigma.\langle t(\sigma), \underline{NIL}\rangle$

In particular:

$$I \triangleq \lambda\sigma.\langle\sigma,\underline{NIL}\rangle$$

The semicolon combinator is defined as:

for t_1 and $t_2 : E$

$$(t_1;t_2) : E$$
$$(t_1;t_2) \triangleq (\lambda\sigma,ido.(ido=\underline{NIL}\rightarrow t_2(\sigma),T\rightarrow\langle\sigma,ido\rangle))^0 t_1$$

The most interesting of the combinators is "$\underline{tixe}$":

for $t_1 : Id \rightarrow E,\quad t_2 : E,\quad p : [Id] \rightarrow Bool$

$$(\underline{tixe}\ [a\rightarrow t_1(a)|p(a)]\ \underline{in}\ t_2) : E$$
$$(\underline{tixe}\ [a\rightarrow t_1(a)|p(a)]\ \underline{in}\ t_2)$$
$$\triangleq \quad (\underline{let}\ e = [a\rightarrow t_1(a)|p(a)]$$
$$\underline{let}\ r(\sigma,ido) = (ido\in\underline{dom}e \rightarrow r^0 e(ido)(\sigma),T\rightarrow\langle\sigma,ido\rangle)$$
$$r^0 t_2)$$

Notice that r is used recursively, thus the effort to resolve an abnormal exit with t_1 continues until p is not satisfied.

Fig 3 provides a rewriting of the exit definition with the above combinator definitions applied to provide a definition in almost-pure lambda notation. Although this is more convenient for the proofs of section 6, the combinators have considerable value in providing a shorter and more intuitive definition of a large language (compare refs Bekić 74 and Allen 72).

Notice that the only labels which are returned from (non-$\underline{NIL}$ second components of the function) "x-l" are those which are not contained in the text argument. Thus it can be proved:

$$pre\text{-}x\text{-}l(ido,nsl) \iff (ido=\underline{NIL} \vee is\text{-}cont(ido,nsl))$$
$$post\text{-}x\text{-}l(ido,nsl,\sigma,\sigma',ido') \iff (ido'=\underline{NIL} \vee \neg is\text{-}cont(ido',nsl))$$

and because of the context condition it is possible to show that x-p is of type:

$$\Sigma \rightarrow \Sigma\ \underline{NIL}$$

and from this extract a denotation of type: $\Sigma \rightarrow \Sigma$

```
x-cp(ido,<nsl>) =
    let e = [id → x-l(id,nsl)|is-cont(id,nsl)]
    let r(σ,ido') = (ido'∈dome → r⁰e(ido')(σ), T → <σ,ido'>)
    r⁰x-l(ido,nsl)


x-l(ido,nsl) =
    ido=NIL              → x-nsl(nsl,1)
    is-dcont(ido,nsl)    → x-nsl(nsl,ind(ido,nsl))
    T                    → (let i = ind(ido,nsl)
                            (λσ,ido'.(ido'=NIL → x-nsl(nsl,i+1)(σ),
                                        T → <σ,ido'>))⁰x-cp(ido,s-b⁰nsl(i))
                           )


x-nsl(nsl,i) =
    i≤lennsl → ((λσ,ido'.(ido'=NIL → x-nsl(nsl,i+1), T → <σ,ido'>))⁰
                    x-s(s-b(nsl(i))))
    T        → λσ.<σ,NIL>


x-g(<id>)  =  λσ.<σ,id>


x-el(<el>)  =  λσ.<el-sem(el)(σ),NIL>


x-p, x-c  unchanged
```

Fig 3: Definition by exit mechanism with combinators expanded

5. DEFINITION BY CONTINUATIONS

This section introduces the more widely used continuation approach for the definition of languages which include goto statements. As with exits, this approach recognises that denotations of type $\Sigma \to \Sigma$ will <u>not</u>

suffice. While continuations themselves are:

$T = \Sigma \rightarrow \Sigma$

the denotations of statement-like constructs become:

$T \rightarrow T$

The question of the meaning of goto statements is handled by associating continuations with identifiers. The denotation of a goto statement for any continuation is then the continuation associated with the contained identifier. In a complex language definition, block structure would anyway force the use of an explicit environment argument to the semantic functions and this can be used to record the associated continuation for labels. Thus, in the current case:

$Env \; = \; Id \rightarrow T$

Intuitively, one can consider statement denotations as yielding, for a given subsequent computation (i.e. continuation), the overall computation starting at this statement. Notice that this is not simply the composition of two functions of type $\Sigma \rightarrow \Sigma$ because of the possibility of "goto". The label denotations are the transitions resulting from starting execution at that label and executing to the end of the program. Thus a function of type $\Sigma \rightarrow \Sigma$ is associated with a text given a particular environment and continuation. A more complete description of the method of continuations is given in *Strachey 74*. The definition by continuations is given in *fig 4*. (The use of braces to bracket arguments which are continuations is adopted for the benefit of the reader.)

Since there are no combinators to be explained in this definition, no intuitive explanation is offered. The reader who is unfamiliar with this style of definition is, however, advised to study this definition carefully (possibly with the aid of an example) to be sure he has grasped the rather back-to-front construction of denotations. The types of these semantic functions are:

$c\text{-}p: \; P \rightarrow T$
$c\text{-}c: \; C \rightarrow (Env \rightarrow (T \rightarrow T))$
$c\text{-}nsl: \; Ns^* \; Nat \rightarrow (Env \rightarrow (T \rightarrow T))$
$c\text{-}s: \; S \rightarrow (Env \rightarrow (T \rightarrow T))$ (assumed)

```
c-p(<cp>)=
     let env0 = [id→c-l(id,s-b(cp))(env0){I} | is-cont(id,s-b(cp))]
     c-c(cp)(env0){I}


c-c(<nsl>)  =  c-nsl(nsl,1)


c-nsl(nsl,i)(env){c}=
    i<lennsl → c-s(s-b(nsl(i)))(env){c-nsl(nsl,i+1)(env){c}}
    T        → c


c-g(<id>)(env){c}  =  env(id)


c-el(<el>)(env){c}  =  c°el-sem(el)


c-l(id,nsl)(env){c}=
    is-dcont(id,nsl) → c-nsl(nsl,ind(id,nsl))(env){c}
    T                → (let i = ind(ido,nsl)
                         c-l(id,s-b°s-b°nsl(i))(env){c-nsl(nsl,i+1)(env){c}}
                        )
```

fig 4: Definition using continuations

$c\text{-}g$: $G \rightarrow (Env \rightarrow (T \rightarrow T))$
$c\text{-}el$: $EL \rightarrow (Env \rightarrow (T \rightarrow T))$
$c\text{-}l$: $Id\ Ns^* \rightarrow (Env \rightarrow (T \rightarrow T))$

6. EQUIVALENCE OF THE TWO DEFINITIONS

Sections 4 and 5 have both provided mappings from programs to functions $(\Sigma \rightarrow \Sigma)$: the aim of this section is to show that the definitions are equivalent in the sense that they associate the same transformation with any well-formed program. It is possible to discern three important differences between the exit and continuation definitions:

(i) The continuation definition associates with each label identifier a denotation (i.e. continuation) which reflects the effect of starting execution at that label and continuing to the end of the entire program. On the other hand, the exit definition provides (see point *(ii)*) different denotations for label identifiers at each nested compound statement: in each case the denotation captures the meaning of execution from any contained label to the end of the current compound statement.

(ii) Whereas the continuation definition passes the denotations of label identifiers to semantic functions explicitly in the environment, the meaning of labels (an E) in exit definitions is used (by the *tixe* combinator) at the level of the containing compound statement.

(iii) The mode of generation of the respective denotations in the two approaches differs: in the exit-style the denotation of a label is derived by starting at that label and "composing" forwards (via the semicolon combinator); continuations are built up from the final transformation composing backwards.

The proof style adopted below is to show a sequence of definitions (each with different prefixes for the function names) and show that each is equivalent to its predecessor. Since the point of departure is the "c-" definition of section 5 and the last step shows the equivalence of the "f-" definition to the (expanded form of the) "x-" definition of section 4 a complete proof of equivalence is given. A good overview of the reasoning can be obtained by understanding the intermediate definitions without following the details of the individual equivalence proofs.

The first step (i.e. the "*d-*" definition) is purely preparatory, as, in a sense, is the second ("*e-*") although this relates specifically to difference *(i)*. The step to the "*f-*" definition completes the resolution of differences *(i)* and *(ii)*. The final step from the "*f-*" to the "*x-*" functions resolves difference *(iii)*.

The first step in our equivalence is trivial. Looking at the "*c-*" functions, it is obvious that *c-c* and *c-l* are both special cases of a more general function which takes an optional identifier as its first argument.

d-l: [*Id*] *Ns** → (*Env* → (*T* → *T*))

Since a combination of these two tasks has been employed in the "*x-*" definition the difference must be resolved somewhere and early resolution will shorten some of the inductive arguments to be used below. In fact the definition given in *fig 5* could have been presented in section 5: equivalence with that actually given follows from:

is-cont(id,nsl) ⇒ *d-l(id,nsl)* = *c-l(id,nsl)*
d-l(NIL,nsl) = *c-c(<nsl>)*

d-c(<nsl>)=d-l(NIL,nsl)

d-l(ido,nsl)(env){c}=
    ido=NIL → *d-nsl(nsl,1)(env){c}*
    is-dcont(ido,nsl)→ *d-nsl(nsl,ind(ido,nsl))(env){c}*
    T → (*let i = ind(ido,nsl)*
        d-l(ido,s-b⁰s-b⁰nsl(i))(env){d-nsl(nsl,i+1)(env){c}}
)

d-p,d-nsl,d-g,d-el: models of respective "*c-*" functions

fig 5: Definition using continuations with merge of *c-c* and *c-l*.

The next step in the proof also changes very little. The types of the "*e-*" functions are the same as those of the "*d-*" functions. The difference is that some elements of the environment (i.e. contained label denotations) are recomputed at each compound statement level. What has to be proved is that the recomputed values are exactly the same as those already stored (the usefulness of this step will become apparent later). A good intuitive confirmation of this claim can be obtained by viewing the "*e-*" functions as a macro-expansion and observing that the continuation argument of *e-cp(NIL,<nsl>)* is identical with that used to generate the denotations (in env_0 of *d-p*) of all labels contained in *nsl*.

Proceeding more formally, from the substitutivity of equal values it is obvious that:

$$\begin{aligned}&(is\text{-}cont(id,nsl) \Rightarrow env(id)=d\text{-}l(id,nsl)(env)\{c\}) \quad \&\\ &(ido=\underline{NIL} \vee is\text{-}cont(ido,nsl))\\ &\quad\Rightarrow d\text{-}l(ido,nsl)(env+[id\mapsto d\text{-}l(id,nsl)(env)\{c\}\mid is\text{-}cont(id,nsl)])\{c\}\\ &\qquad = d\text{-}l(ido,nsl)(env)\{c\}\end{aligned}$$

It is now necessary to show that for all

$$d\text{-}l(ido,nsl)(env)\{c\}$$

it is true that:

$$is\text{-}cont(id,nsl) \Rightarrow env(id) = d\text{-}l(id,nsl)(env)\{c\}$$

Observe that this is true for the reference to *d-l* from *d-p*. For recursive calls of *d-l* consider:

id_n such that $is\text{-}cont(id_n,nsl)\ \&\ \neg is\text{-}dcont(id_n,nsl)$

its denotation is given by:

$$env(id_n) = d\text{-}l(id_n,s\text{-}b^0\ s\text{-}b^0 nsl(i_n))(env)\{d\text{-}nsl(nsl,i_n+1)(env)\{c\}\}$$
where $i_n = ind(id_n,nsl)$

but for recursive references to *d-l* in $d\text{-}nsl(nsl,i_n)(env)\{c\}$

$$d\text{-}nsl(nsl,i_n)(env)\{c\}$$
$$= d\text{-}s(s\text{-}b^0 nsl(i_n))(env)\{d\text{-}nsl(nsl,i_n+1)(env)\{c\}\}$$
$$= d\text{-}l(\underline{NIL}, s\text{-}b^0 s\text{-}b^0 nsl(i_n))(env)\{d\text{-}nsl(nsl,i_n+1)(env)\{c\}\}$$

so for:

$$is\text{-}cont(id_n, s\text{-}b^0 s\text{-}b^0 nsl(i_n))$$

the required property still holds since the continuations match. This concludes the argument and the definition in *fig 6* can be seen to be equivalent to the "*d-*" functions because *e-cp* is introduced just to "recompute" some label denotations; other functions are changed accordingly including the fact that *e-p* need no longer generate an environment:

$$e\text{-}cp : [Id]\ C \rightarrow (Env \rightarrow (T \rightarrow T))$$

The next stage of the proof is the most interesting. Before coming to the "*f-*" functions a useful lemma on continuations will be given. Intuitively this lemma states that in order to achieve the same effect as composing some function with the denotation of a statement, that function must be composed with both the contination and each label denotation used in deriving the given denotation.

```
e-p(<cp>)  =  e-c(cp)([]){I}


e-c(cp)  =  e-cp(NIL,cp)


e-cp(ido,<nsl>)(env){c}=
  let env' = env+[id→e-l(id,nsl)(env'){c}|is-cont(id,nsl)]
  e-l(ido,nsl)(env'){c}


e-l(ido,nsl)(env){c}=
  ido=NIL               →  e-nsl(nsl,1)(env){c}
  is-dcont(ido,nsl)     →  e-nsl(nsl,ind(ido,nsl))(env){c}
  T                     →  (let i = ind(ido,nsl)
                             e-cp(ido,s-b^0nsl(i))(env){e-nsl(nsl,i+1)(env){c}}
                           )
```

e-nsl,e-g,e-el: models of respective "*c-*" functions

fig 6: Definition using continuations recomputed at each compound statement

Lemma I

define: $me(c,env) = [id \to c^{0} env(id) \mid id \in \underline{dom}\, env]$

show for: et is $e\text{-}s(s), e\text{-}nsl(nsl,i), e\text{-}l(ido,nsl)$ or $e\text{-}cp(ido,cp)$

that: $c_2{}^{0} et(env)\{c_1\} = et(me(c_2,env))\{c_2{}^{0} c_1\}$

Proof:

The argument is by induction on the structure of the text, as a basis consider statements of G and E:

$$c_2{}^{0} e\text{-}g(<id>)(env)\{c\} = c_2{}^{0} env(id)$$
$$e\text{-}g(<id>)(me(c_2,env))\{c_2{}^{0} c_1\} = me(c_2,env)(id)$$
$$= c_2{}^{0} env(id)$$

$$c_2{}^{0} e\text{-}el(<el>)(env)\{c_1\} = c_2{}^{0} c_1{}^{0} el\text{-}sem(el)$$
$$e\text{-}el(<el>)(me(c_2,env))\{c_2{}^{0} c_1\} = c_2{}^{0} c_1{}^{0} el\text{-}sem(el)$$

next in the basis consider elements of Ns^* where no element contains a C, here a subsidiary inductive proof (on $\underline{len}\, nsl-i$) is made. For the basis, consider $i > \underline{len}\, nsl$:

$$c_2{}^{0} e\text{-}nsl(nsl,i)(env)\{c_1\} = c_2{}^{0} c_1$$
$$e\text{-}nsl(nsl,i)(me(c_2,env))\{c_2{}^{0} c_1\} = c_2{}^{0} c_1$$

for the inductive step $i \le \underline{len}\, nsl$:

$$c_2{}^{0} e\text{-}nsl(nsl,i)(env)\{c_1\}$$
$$= c_2{}^{0} e\text{-}s(s\text{-}b(nsl(i)))(env)\{e\text{-}nsl(nsl,i+1)(env)\{c_1\}\}$$
$$= e\text{-}s(s\text{-}b(nsl(i)))(me(c_2,env))\{c_2{}^{0} e\text{-}nsl(nsl,i+1)(env)\{c_1\}\} \qquad \text{I.H. on } S$$
$$= e\text{-}s(s\text{-}b(nsl(i)))(me(c_2,env))\{e\text{-}nsl(nsl,i+1)(me(c_2,env))\{c_2{}^{0} c_1\}\} \qquad \text{I.H. on } Ns^*$$

$$e\text{-}nsl(nsl,i)(me(c_2,env))\{c_2{}^{0} c_1\}\}$$
$$= e\text{-}s(s\text{-}b(nsl(i)))(me(c_2,env))\{e\text{-}nsl(nsl,i+1)(me(c_2,env)\{ c_2{}^{0} c_1\}\}$$

For elements of Ns^* where no element contains a compound statement, the results for $e\text{-}l(ido,nsl)$ and $e\text{-}cp(ido,cp)$ are immediate from the above.

For the inductive step, the only additional case to be considered is the construction of elements of C, thus:

$$c_2 {}^{0} e\text{-}c(cp)(env)\{c_1\} = c_2 {}^{0} e\text{-}cp(s\text{-}b(cp))(env)\{c_1\}$$
$$e\text{-}c(cp)(me(c_2,env))\{c_2 {}^{0} c_1\} = e\text{-}cp\,(s\text{-}b(cp))(me(c_2,env))\{c_2 {}^{0} c_1\}$$

which are equal by induction hypothesis.

This concludes the proof of Lemma I.

Lemma I will now be used to justify change from passing in label denotations in environments to composing them with the revised meaning of the basic statement list. The revision to the meaning of a statement list changes it to type E and makes any goto statement cause a label to be returned as the second component of the result. The composition of the label denotations is now (recursively) applied only if this indication of abnormal exit is present. Intuitively the proof which follows shows that any environment is equivalent to a composition of a test and a constant environment, and any continuation is equivalent to a composition of a test and a constant function. Since both of these tests are the same, lemma I can be used to factor out the test.

Proceeding more formally, it is observed that though the used types of the "*e-*" functions are:

$$e\text{-}\theta: \quad \Theta \rightarrow ((Id \rightarrow T) \rightarrow (T \rightarrow T))$$

they are perfectly general in that they also fit:

$$e\text{-}\theta: \quad \Theta \rightarrow ((Id \rightarrow (\Sigma \rightarrow \Omega)) \rightarrow ((\Sigma \rightarrow \Omega) \rightarrow (\Sigma \rightarrow \Omega)))$$

Writing:

$$xe(nsl) = [id \rightarrow \lambda\sigma.\langle\sigma,id\rangle \,|\, is\text{-}cont(id,nsl)]$$
$$xt(env,c) = \lambda\sigma,a.(a=\underline{NIL}\rightarrow c(\sigma), T\rightarrow env(a)(\sigma))$$

it is immediate that:

$$\underline{dom}\,env = \{id \,|\, is\text{-}cont(id,nsl)\}$$
$$\Rightarrow \quad [id\rightarrow xt(env,c)^{0} xe(nsl) \,|\, id\in\underline{dom}\,env] = env$$

and:

$$xt(env,c)^{0} \lambda\sigma.\langle\sigma,NIL\rangle = c$$

But then:

$$e\text{-}l(ido,nsl)(env')\{c\} = e\text{-}l(ido,nsl)([id \to xt(env',c)^{0} xe(nsl) \mid id \in \underline{dom}\,env'])\{xt(env',c)^{0} \lambda\sigma.<\sigma,\underline{NIL}>\}$$
$$= xt(env',c)^{0} e\text{-}l(ido,nsl)(xe(nsl))\{\lambda\sigma.<\sigma,\underline{NIL}>\}$$

so:

$$e\text{-}cp(ido,<nsl>)(env)\{c\}$$
$$= (\underline{let}\ env'=env+ [id \to (\lambda\sigma,a.(a=\underline{NIL} \to c(\sigma), T \to env'(a)(\sigma)))^{0}\ e\text{-}l(id\,,\,nsl)(xe(nsl))\{\lambda\sigma.<\sigma,\underline{NIL}>\} \mid is\text{-}cont(id,nsl)]$$
$$(\lambda\sigma,a.(a=\underline{NIL} \to c(\sigma), T \to env'(a)(\sigma)))^{0}\ e\text{-}l(ido,nsl)(xe(nsl))\{\lambda\sigma.<\sigma,\underline{NIL}>\})$$
$$= (\underline{let}\ e=[id \to e\text{-}l(id,nsl)(xe(nsl))\{\lambda\sigma.<\sigma,\underline{NIL}>\} \mid is\text{-}cont(id,nsl)]$$
$$\underline{let}\ r(\sigma,a) = (a \in \underline{dom}\,e \to r^{0} e(a)(\sigma), T \to env(a)(\sigma))$$
$$r^{0} e\text{-}l(ido,nsl)(xe(nsl))\{\lambda\sigma.<\sigma,\underline{NIL}>\})$$

Strictly, the whole definition has now become:

$$e\text{-}p: \quad P \to (\Sigma \to \Sigma\ [Id])$$

(and this was why it was necessary to observe above that T could be replaced by $\Sigma \to \Omega$). But, as with the exit definition in section 4, it can be shown that only non-contained labels can be returned. Thus at the program level it can be shown for well-formed programs that the second element of the result must be $\underline{NIL}$.

But all *env* arguments now give constant denotations for labels! Because the definition only considers well-formed programs these constant functions can be moved into the semantic definition of goto. Furthermore, since the environment argument is now used nowhere, it can be omitted. This results in the definition in *fig 7*.

```
f-p(<cp>) = f-c(cp){λσ.<σ,NIL>}


f-c(cp) = f-cp(NIL,cp)


f-cp(ido,<nsl>){c}=
   let e = [id→f-l(id,nsl){λσ.<σ,NIL>}|is-cont(id,nsl)]
   let r(σ,a) = (a∈dome→r^0e(a)(σ),T→λσ.<σ,a>)
   r^0f-l(ido,nsl){λσ.<σ,NIL>}


f-l(ido,nsl){c}=
   ido=NIL              → f-nsl(nsl,1){c}
   is-dcont(ido,nsl)    → f-nsl(nsl,ind(ido,nsl)){c}
   T                    → (let i = ind(ido,nsl)
                             f-cp(ido,s-b^0nsl(i)){f-nsl(nsl,i+1){c}}
                          )


f-nsl(nsl,i){c}=
   i≤lennsl   →  f-s(s-b^0nsl(i)){f-nsl(nsl,i+1){c}}
   T          →  c


f-g(<id>){c}  =  λσ.<σ,id>


f-el(<el>){c}  =  c^0el-sem(el)
```

fig 7: Definition using (*E*) continuations without environments.

The "*f-*" functions have the types:

```
f-p:    P          → E
f-c:    C          → (E → E)
f-cp:   [Id] C     → (E → E)
e:      Id         → E
```

r:	Σ [*Id*]	→ *E*	
f-l:	[*Id*] *Ns**	→ (*E* → *E*)	
f-nsl:	*Ns** *Nat*	→ (*E* → *E*)	
f-s:	*S*	→ (*E* → *E*)	(assumed)

This definition presents one in which the earlier differences *(i)* and *(ii)* have been eliminated and which only requires the equivalence of the alternative directions for computing denotations to be established to complete the equivalence proof to the "*x-*" functions of section 4.

The approach to this last difference is similar to that taken at the previous stage. Firstly a lemma is introduced which shows that the "*f-*" functions are equivalent to a composition of a test and the corresponding "*x-*" function. Whereas in the previous stage the test was a simulation of the "<u>*tixe*</u>" combinator, this stage is simulating the "*;*" combinator. Applying this lemma generates a set of functions (which could be written out as "*g-*" functions) which pass the same constant "continuation" of $\lambda\sigma.<\sigma,\underline{NIL}>$ to all functions. Once again this constant can be dropped and written directly in the two places where the argument had previously been used. We then have precisely the expanded form of the "*x-*" functions from section 4.

Formally, the lemma is

<u>Lemma II</u>

define: $t(c) = \lambda\sigma,a.(a=\underline{NIL} \rightarrow c(\sigma), T \rightarrow <\sigma,a>)$

show for: $ft(xt)$ is $f\text{-}s(x\text{-}s), f\text{-}nsl(x\text{-}nsl)$ or $f\text{-}l(x\text{-}l)$ respectively

that: $t(c)^{0}xt(s) = ft(s)\{c\}$

the proof (not given here) is by a similar induction to that of Lemma I.

Using Lemma II:

$$f\text{-}nsl(nsl,i)\{c\}=$$
$$i \leq \underline{len}nsl \rightarrow t(f\text{-}nsl(nsl,i+1)\{c\})^{0}x\text{-}s(s\text{-}b^{0}nsl(i))$$
$$T \rightarrow c$$

and:

$f\text{-}l(ido,nsl)\{c\}=$

...

$T \rightarrow (let\ i = ind(ido,nsl)$
$t(f\text{-}nsl(nsl,i+1)\{c\})^0 x\text{-}cp(ido,s\text{-}b^0 nsl(i)))$

rewriting the second case of *f-nsl* as:

$\lambda\sigma.<\sigma,\underline{NIL}>$

and the definition of *f-el* as

$\lambda\sigma.<el\text{-}sem(el)(\sigma),\underline{NIL}>$

the continuation arguments to all functions can be dropped and the "*x-*" functions remain.

7. DISCUSSION

Two different definitions of a language have been given and proved equivalent. It is important to realize that this is a limited proof in the sense that nothing has been established about the power of the two mechanisms in general. In fact continuations can be stored and passed in a way which cannot be simulated by exit. Thus, co-routines or like features can be defined using continuations but not exits (see Reynolds 74). However, both approaches have been used to define major programming languages (cf. Mosses 74, Bekić 74, Henhapl 78) and there is experience from the work on abstract interpreter definitions to argue that where a more powerful construct is not necessary, its use should be avoided.

The choice of which technique is most appropriate might well depend on the intended use of a definition. For general clarity it could be that the ability of the exit combinators to hide the effect of a goto in most parts of a language definition is valuable. On the other hand, proofs about the meaning of programs will anyway have to expose the combinators and a continuation definition may be more directly usable. Even here there is one important advantage of the exit approach and that is the ability to localize the effect of goto statements within the syntactic unit containing the goto and the label. Thus in:

```
begin
   .
   .
   .
   begin
      .
      .
      .
      begin
         .
         .
         .
         goto ℓ,
         .
         .
         .
      end;
      .
      .
      .
      ℓ: ... ;
      .
      .
      .
   end
   .
   .
   .
end
```

the second nested block will have a denotation of type:

$$\Sigma \rightarrow \Sigma \ \underline{NIL}$$

This closing-off of the semantic effects of goto cannot be simulated with continuations.

Both the Oxford and Vienna groups have made experiments with using definitions to provide a starting point for systematic (justified) compiler development (see Milne 76, Jones 76a). It is in this area that a more meaningful comparison of continuations and exits should be sought.

Hopefully the proof in section 6 has been presented in an intuitively clear style. For more interesting approaches to such proofs see Reynolds 74, Reynolds 75.

ACKNOWLEDGEMENTS

An earlier draft of this paper was submitted to, and eventually accepted by the IBM Journal of Research and Development. Two of the referees provided very valuable comments for which the author would like to express his gratitude. (The paper was subsequently withdrawn with the agreement of the editor of that journal in order to place it in the current volume.)