A semantics for while with break, continue and goto

Patricia Peratto
psperatto@adinet.com.uy
Uruguay

August 2006

Abstract

This work presents a formal description of a subset of a **C-like** language in the form of an *operational semantics*. We give semantics to the following statements (presented in alphabetical order) : *assignation*, **break**, *composition*, **continue**, **goto**, **if**, **skip** and **while**. The semantics is given by an abstract machine composed by an stack, two counters and three functions. We prove some expected properties of the semantics.

1 Introduction

There is few work concerning the *operational semantics* of statements **break**, **continue** and **goto** for **C**-like languages. Previous work does not treat these statements or take a different point of view. In particular the sentence **goto** was not treated at all.

The two counters of our abstract machine index **while**-loops. Using them together with the second function, we can know when an instruction **break**, **continue** or **goto** is inside an *active* **while**-loop (we say a **while**-loop is *active* when its boolean condition evaluates to *true*).

The stack works as a continuation, we put in it the sentences that have to be executed after the actual one.

The first of the functions maps *labels* to *environments* and using it we give meaning to **goto** statements.

The third function represents an *state* in which we store the values of program variables.

The outline of our paper is as follows: section 2 presents the abstract syntax of the language (this includes arithmetic and boolean expressions and

statements). Section 3 presents the semantics of expressions and section 4 the semantics of statements. We prove the semantics follows the intended meaning of the statements **continue**,**break** and **goto**. In section 5 we prove the semantics is deterministic. In section 6 we present related work and in section 7 conclusions.

2 Abstract syntax of the language

We use a syntactic notation based on BNF. Parenthesis can be used (not indicated in our BNF) to solve ambiguities and uniquely determine the corresponding parse tree.

We have the following Syntactic Categories and meta-variables ranging over them

n will range over numerals, **Num**
a will range over arithmetic expressions, **AExp**
x will range over arithmetic variables, **AVar**
b will range over boolean expressions, **BExp**
x will range over boolean variables, **BVar**
S will range over statements, **Stm**

The meta-variables can be primed or subscripted for example a, a', a_1, a_2 all stand for arithmetic expressions. x refers both to arithmetic and boolean variables. What we mean can be inferred from the context.

Definition 2.1. *Abstract syntax for Arithmetic Expressions*

$$a ::= n \mid x \mid a_1 + a_2 \mid a_1 * a_2 \mid a_1 - a_2$$

Definition 2.2. *Abstract syntax for Boolean Expressions*

$$b ::= \textbf{true} \mid \textbf{false} \mid x \mid a_1 = a_2 \mid a_1 < a_2 \mid \neg b \mid b_1 \vee b_2 \mid b_1 \wedge b_2$$

true and **false** stand for constant truth values.

Definition 2.3. *Abstract syntax for Statements*

$S ::= x := a \mid x := b \mid \textbf{skip} \mid S_1; S_2 \mid \textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2 \mid$
$\textbf{while } b \textbf{ do } S \mid \textbf{continue} \mid \textbf{break} \mid label : S \mid \textbf{goto } label$

label stands for an identifier.

3 Semantics of expressions

We define the following semantic functions:

$A : \textbf{Aexp} \to (\textbf{State} \to \textbf{Z})$ for arithmetic expressions
$B : \textbf{Bexp} \to (\textbf{State} \to \textbf{T})$ for boolean expressions

where

Z is the semantic domain of integers ranged over by metavariables n, z,
T is the semantic domain of the truth values ranged over by metavariable t

and

$\textbf{State} = (\textbf{AVar} + \textbf{BVar}) \to (\textbf{Z} + \textbf{T})$

is a semantic function (from syntactic domains to semantic domains) ranged over by s.
$+$ denotes disjoint union, with constructors *inl* and *inr* both over syntactic domains and over semantic domains. $s \in \textbf{State}$ satisfies $s(inl\ x) \in \textbf{Z}$ and $s(inr\ x) \in \textbf{T}$.

Definition 3.1. *Semantics of Arithmetic Expressions*
The transition relation is specified by the rules of the transition system presented in Table 1.

$+, *, -$ at the left denote syntax and at the right the corresponding operations over the semantical domains.
We define the semantic function

$A(a)s = z$ if and only if $\langle a, s\rangle \to z$

Theorem 3.2. *A* **is a function**
Given a and s, there exists only one z such that $\langle a, s\rangle \to z$.

$$\langle n, s\rangle \to n$$

$$\langle x, s\rangle \to s(inl\ x)$$

$$\frac{\langle a_1, s\rangle \to z_1 \quad \langle a_2, s\rangle \to z_2}{\langle a_1 + a_2, s\rangle \to z_1 + z_2}$$

$$\frac{\langle a_1, s\rangle \to z_1 \quad \langle a_2, s\rangle \to z_2}{\langle a_1 * a_2, s\rangle \to z_1 * z_2}$$

$$\frac{\langle a_1, s\rangle \to z_1 \quad \langle a_2, s\rangle \to z_2}{\langle a_1 - a_2, s\rangle \to z_1 - z_2}$$

Table 1: Semantics of arithmetic expressions

$$\langle \mathbf{true}, s\rangle \to \mathbf{true}$$

$$\langle \mathbf{false}, s\rangle \to \mathbf{false}$$

$$\langle x, s\rangle \to s(inr\ x)$$

$$\frac{A(a_1)s = z_1 \quad A(a_2)s = z_2}{\langle a_1 = a_2, s\rangle \to z_1 = z_2}$$

$$\frac{A(a_1)s = z_1 \quad A(a_2)s = z_2}{\langle a_1 < a_2, s\rangle \to z_1 < z_2}$$

$$\frac{\langle b, s\rangle \to t}{\langle \neg b, s\rangle \to \neg t}$$

$$\frac{\langle b_1, s\rangle \to t_1 \quad \langle b_2, s\rangle \to t_2}{\langle b_1 \vee b_2, s\rangle \to t_1 \vee t_2}$$

$$\frac{\langle b_1, s\rangle \to t_1 \quad \langle b_2, s\rangle \to t_2}{\langle b_1 \wedge b_2, s\rangle \to t_1 \wedge t_2}$$

Table 2: Semantics of boolean expressions

Proof. Structural Induction on a. □

Definition 3.3. *Semantics of Boolean Expressions*
Table 2 presents the corresponding transition system

$=, <, \neg, \vee, \wedge$ at the left denotes syntax and at the right the corresponding operations over the semantical domains.

We define the semantic function

$B(b)s = t$ if and only if $\langle b, s\rangle \rightarrow t$

Theorem 3.4. B **is a function**
Given b and s, there exists only one t such that $\langle b, s\rangle \rightarrow t$.

Proof. Structural Induction on b. □

4 Semantics of Statements

We use pattern matching and conditionals to specify the abstract machine. Conditionals supplement pattern matching allowing to impose conditions over pattern variables as well as to check the values of functions.

The way of reading our pattern matching definition is from top to bottom. The patterns are allowed to overlap (a variable overlaps all the patterns). When a pattern does not match, we continue with the following pattern.

Our pattern matching is exhaustive, this is achieved with variables in the pattern as the default case.

We specify a transition system with configurations of the form

$$\langle S, n, y, E, \rho, \gamma, s\rangle.$$

S is the corresponding statement, n is the first counter, y the second counter (both n and y are natural numbers).
E is an environment of pairs $(statement, counter)$ typed as:

$$E : \mathbf{Stack}(Stm \times N)$$

where *Stack* has constructors : and *nil*.
ρ is the first function that has as domain the set of *labels* and as codomain the set of *environments*.
γ is the second function used to know when a **while** statement is active.

We denote *States* by s. We specify by $s[x := n]$ the state such that $(s[x := n]\ x) = n$ and $(s[x := n]\ y) = (s\ y)$.

Table 3 presents the semantics of statements.

4.1 Use of the counters

We use the counters to number **while**-loops.

The first counter is a number associated to the statement when is being executed, the second counter equals the first natural number not assigned yet as first counter to a **while** statement.

When we have a **while** statement in the top of the environment occur one of the following two cases

1. the **while**-loop is active and was put in the stack to repeat the iteration. In this case its first counter does not change.

2. The **while** statement is not active. We assign to it as first counter the second counter.

We use the first counter also in relation to the statements **continue** and **break**.

When we find an active **while** statement, we have to execute its body and after repeat the execution of the statement, for this the body is put as the sentence at the top of the environment and the sentence **while** is put below it.

We associate as counter to the **while** statement in the environment the value of its first counter, and as counter for its body its successor. This will be used in relationship with statements **continue** and **break** and is explained afterwards when we explain the intended meaning of statements.

4.2 Use of the functions ρ and γ

The first function (ρ) stores for each *label*, an environment. When we meet a **goto** *label* statement, we continue the execution in the environment aso-

ciated to *label*.

The statement *label*:S can be found "before" than a corresponding **goto**; in this case, when we reach the **goto** statement the function ρ has an environment as definition for the corresponding *label*.

If this is not the case, the definition for the corresponding environment is done after finding the **goto** *label* statement. We go forward with the statements in the current environment until finding the statement labeled *label*. This statement and the ones in the current environment form the environment for *label*.

The second function (γ) is used to know if a **while** statement is active. Is defined in such a way that $\gamma n = true$ if the **while** statement numbered n is active and $\gamma n = false$ otherwise.

4.3 The auxiliar function h

When we meet an **if** instruction, we choose the **then** or **else** part depending on the boolean condition. The corresponding sentence is put in the environment as first sentence to be executed.

We always put the next sentence in the top of the environment because in case of being a **while** instruction we give a special treatment (that can be seen in the management of the indexes).

Can be the case of a labeled statement appearing in the **then** or **else** part of an **if** statement. To define the corresponding environment in the function ρ we define the function h, that appears at the end of Table 3.

We use the function h also in case of a not active **while** statement. Since when an active **while** is find, this statement is put in the stack, any *label* found in the **while**'s body will have the **while** statement in its associated environment. In case the **while** statement is defused after, we have to associate to any *label* find in its body, an environment without this **while** statement. For this, we "compute" again the environment when we reach the defused while statement, applying function h.

$$\langle x := a, n, y, E, \rho, \gamma, s\rangle \rightarrow \langle \mathbf{skip}, n, y, E, \rho, \gamma, s[inl\ ax := inl\ A(a)s]\rangle$$

$$\langle x := b, n, y, E, \rho, \gamma, s\rangle \rightarrow \langle \mathbf{skip}, n, y, E, \rho, \gamma, s[inr\ bx := inr\ B(b)s]\rangle$$

$$\langle \mathbf{skip}, m, y, (S_1; S_2, n) : E, \rho, \gamma, s\rangle \rightarrow \langle \mathbf{skip}, m, y, (S_1, n) : (S_2, n) : E, \rho, \gamma, s\rangle$$

$$\langle \mathbf{skip}, m, y, (\mathbf{while}\ b\ \mathbf{do}\ S, n) : E, \rho, \gamma, s\rangle \rightarrow \langle \mathbf{while}\ b\ \mathbf{do}\ S, n, y, E, \rho, \gamma, s\rangle \qquad \gamma\ n\ = true$$

$$\langle \mathbf{skip}, m, y, (\mathbf{while}\ b\ \mathbf{do}\ S, n) : E, \rho, \gamma, s\rangle \rightarrow \langle \mathbf{while}\ b\ \mathbf{do}\ S, y, s(y), E, \rho, \gamma, s\rangle$$

$$\langle \mathbf{skip}, m, y, (S, n) : E, \rho, \gamma, s\rangle \rightarrow \langle S, n, E, \rho, \gamma, s\rangle$$

$$\langle \mathbf{skip}, n, y, nil, \rho, \gamma, s\rangle \rightarrow s$$

$$\langle label : S, n, y, E, \rho, \gamma, s\rangle \rightarrow \langle \mathbf{skip}, n, y, (S, n) : E, \lambda z.if\ (z = label)\ then\ (S, n) : E\ else\ (\rho\ z), \gamma, s\rangle$$

$$\langle \mathbf{if}\ b\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2, n, y, E, \rho, \gamma, s\rangle \rightarrow^{b=true} \langle \mathbf{skip}, n, y, (S_1, n) : E, h(S_1, n, E, h(S_2, n, E, \rho)), \gamma, s\rangle$$

$$\langle \mathbf{if}\ b\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2, n, y, E, \rho, \gamma, s\rangle \rightarrow^{b=false} \langle \mathbf{skip}, n, y, (S_2, n) : E, h(S_1, n, E, h(S_2, n, E, \rho)), \gamma, s\rangle$$

Table 3 : Semantics of Statements

4.4 The semantics and the intended meaning of statements

We show that the transition rules of the semantics of statements **continue**, **break** and **goto** follow their intended meaning.

4.4.1 The meaning of continue

If **continue** is inside an active **while**-loop has a counter of the form $s(n)$. In this case $\gamma\ n = true$. We have to skip all the sentences with the same number than **continue** (are sentences inside the same **while**-loop). We skip all these sentences until finding the **while** sentence with index n. This is achieved by the following transitions:

$$\langle \textbf{continue}, s(m), y, (S, s(n)) : E, \rho, \gamma, s\rangle \rightarrow \langle \textbf{continue}, s(m), y, E, \rho, \gamma, s\rangle \quad \gamma\ m = true \wedge n = m$$

$$\langle \textbf{continue}, s(n), y, (\textbf{while}\ b\ \textbf{do}\ S, n) : E, \rho, \gamma, s\rangle \rightarrow \langle \textbf{while}\ b\ \textbf{do}\ S, n, y, E, \rho, \gamma, s\rangle \quad \gamma\ n = true$$

If **continue** is at the greater level of the program (outside **while** loops) or inside a not active **while** loop its counter is 0, so the first two clauses does not match and the clause applyied is one of the following

$$\langle \textbf{continue}, m, y, (S, n) : E, \rho, \gamma, s\rangle \rightarrow \langle \textbf{skip}, m, y, (S, n) : E, \rho, \gamma, s\rangle$$

$$\langle \textbf{continue}, n, y, nil, \rho, \gamma, s\rangle \rightarrow s$$

this is, behaves as skip, i.e. continues with the next sentence, and finishes with the execution in the case there are no more sentences in the program.

4.4.2 The meaning of break

If **break** is inside an active **while**-loop, has a counter of the form $s(n)$ where $\gamma\ n = true$. We have to skip all the sentences in the environment until finding the corresponding **while** sentence. This is achieved with the rules

$$\langle \textbf{while } b \textbf{ do } S, n, y, nil, \rho, \gamma, s\rangle \rightarrow^{b=false} s$$

$$\langle \textbf{while } b \textbf{ do } S, m, y, E, \rho, \gamma, s\rangle \rightarrow^{b=false} \langle\ \textbf{skip}, m, E, h(S, 0, E, \rho), \lambda z.if\ (z = m)\ then\ false\ else\ (\gamma\ z), s\rangle$$

$$\langle \textbf{while } b \textbf{ do } S, n, E, \rho, \gamma, s\rangle \rightarrow^{b=true} \langle\ \textbf{skip}, 0, (S, s(n)) : (\textbf{while } b \textbf{ do } S : n) : E, \rho, \lambda z.if\ (z = n)\ then\ true\ else\ (\gamma\ z), s\rangle$$

$$\langle \textbf{continue}, s(m), y, (S, s(n)) : E, \rho, \gamma, s\rangle \rightarrow \langle \textbf{continue}, s(m), y, E, \rho, \gamma, s\rangle \qquad \gamma\ m = true \wedge n = m$$

$$\langle \textbf{continue}, s(n), y, (\textbf{while } b \textbf{ do } S, n) : E, \rho, \gamma, s\rangle \rightarrow \langle \textbf{while } b \textbf{ do } S, n, y, E, \rho, \gamma, s\rangle \qquad \gamma\ n = true$$

$$\langle \textbf{continue}, m, y, (S, n) : E, \rho, \gamma, s\rangle \rightarrow \langle \textbf{skip}, m, y, (S, n) : E, \rho, \gamma, s\rangle$$

$$\langle \textbf{continue}, n, y, nil, \rho, \gamma, s\rangle \rightarrow s$$

$$\langle \textbf{break}, s(m), y, (S, s(n)) : E, \rho, \gamma, s\rangle \rightarrow \langle \textbf{break}, s(m), y, E, \rho, \gamma, s\rangle \qquad \gamma\ m = true \wedge n = m$$

$$\langle \textbf{break}, s(n), y, (\textbf{while } b \textbf{ do } S, n) : E, \rho, \gamma, s\rangle \rightarrow \langle\ \textbf{skip}, n, y, E, \rho, \lambda\ z.\ if\ (z = n)\ then\ false\ else\ (\gamma\ z), s\rangle \qquad \gamma\ n\ =\ true$$

$$\langle \textbf{break}, m, y, (S, n) : E, \rho, \gamma, s\rangle \rightarrow \langle \textbf{skip}, n, y, (S, n) : E, \rho, \gamma, s\rangle$$

$$\langle \textbf{break}, n, y, nil, \rho, \gamma, s\rangle \rightarrow s$$

Table 3 : Semantics of Statements (continuation (a))

$$\langle \textbf{goto}\ label, m, y, E, \rho, \gamma, s\rangle \rightarrow \langle \textbf{skip}, m, y, (\rho\ label), \rho, \gamma, s\rangle \qquad (\rho\ label) \neq nil$$

$$\langle \textbf{goto}\ label, m, y, (S_1; S_2, n) : E, \rho, \gamma, s\rangle \rightarrow \langle \textbf{goto}\ label, m, y, (S_1, n) : (S_2, n) : E, \rho, \gamma, s\rangle$$

$$\langle \textbf{goto}\ label, m, y, (label' : S, n) : E, \rho, \gamma, s\rangle \rightarrow \langle\ \textbf{goto}\ label, m, y, E, \lambda\ z.\ if\ (z = label')\ then\ (S, n) : E\ else\ (\rho\ z), \gamma, s\rangle$$

$$\langle \textbf{goto}\ label, m, y, (label\ S, n) : E, \rho, \gamma, s\rangle \rightarrow \langle\ \textbf{skip}, n, y, (S, n) : E, \lambda z.if\ (z = label)\ then\ (S, n) : E\ else\ (\rho\ z), \gamma, s\rangle$$

$$\langle \textbf{goto}\ label, m, y, (\textbf{while}\ b\ \textbf{do}\ S, n) : E, \rho, \gamma, s\rangle \rightarrow \langle\ \textbf{goto}\ label, m, y, E, \rho, \lambda\ z.\ if\ (z = n)\ then\ false\ else\ (\gamma\ z), s\rangle \qquad (\gamma\ n)\ = true \wedge s(n) \leq m$$

Table 3: Semantics of Statements (continuation (b))

$$\langle \textbf{goto}\ label, m, y, (\textbf{while}\ b\ \textbf{do}\ S, n) : E, \rho, \gamma, s\rangle \rightarrow \langle\ \begin{array}{l}\textbf{goto}\ label,\\ m,\\ y,\\ (S,0) : E,\\ \rho,\\ \lambda\ z.\ if\ (z = n)\ then\ false\ else\ (\gamma\ z),\\ s\rangle\end{array}$$

$$\langle \textbf{goto}\ label, m, y, (\textbf{if}\ b\ \textbf{then}\ S_1\ \textbf{else}\ S_2, n) : E, \rho, \gamma, s\rangle \rightarrow \langle\ \begin{array}{l}\textbf{goto}\ label,\\ m,\\ y,\\ E,\\ h(S_1, n, E, h(S_2, n, E, \rho),\\ \gamma,\\ s\rangle\end{array}$$

$$\langle \textbf{goto}\ label, m, y, (S, n) : E, \rho, \gamma, s\rangle \rightarrow \langle \textbf{goto}\ label, m, y, E, \rho, \gamma, s\rangle$$

$$h(label : S_1; S_2, n, E, \rho) \ =\ \lambda\ z\ .\ if\ (z = label)\ then\ (S_1; S_2, n) : E \ else\ h(S_2, n, E, \rho)$$

$$h(label : S, n, E, \rho) \ =\ \lambda\ z\ .\ if\ (z = label)\ then\ (S, n) : E \ else\ \rho$$

$$h(S_1; S_2, n, E, \rho) = h(S_2, n, E, \rho)$$

$$h(S, n, E, \rho) \ =\ \rho$$

Table 3 : Semantics of Statements (continuation (c))

$$\langle \textbf{break}, s(m), y, (S, s(n)) : E, \rho, \gamma, s\rangle \rightarrow \langle \textbf{break}, s(m), y, E, \rho, \gamma, s\rangle \quad \gamma\ m = true \wedge n = m$$

$$\langle \textbf{break}, s(n), y, (\textbf{while}\ b\ \textbf{do}\ S, n) : E, \rho, \gamma, s\rangle \rightarrow \langle\ \textbf{skip}, n, y, E, \rho, \lambda\ z.\ if\ (z = n)\ then\ false\ else\ (\gamma\ z), s\rangle \quad \gamma\ n\ =\ true$$

If **break** is at the greater level of the program (outside **while**-loops) or inside a not active **while** loop its counter is 0, so the first two clauses does not match and the clause applyied is one of the following

$$\langle \textbf{break}, m, y, (S, n) : E, \rho, \gamma, s\rangle \rightarrow \langle \textbf{skip}, m, y, (S, n) : E, \rho, \gamma, s\rangle$$

$$\langle \textbf{break}, n, y, nil, \rho, \gamma, s\rangle \rightarrow s$$

break outside a **while**-loop behaves as skip.

The reason while we control in the transitions for **continue** and **break** if $\gamma\ n = true$ is because we can find a sentence **continue** or **break** after a *jump* to a sentence inside a **while**-loop. The behaviour of the **continue** or **break** sentence depends on the **while** being active or not. In case is not active, we have not to skip sentences and we continue with the next statement below it.

4.4.3 The meaning of goto

If the sentence *label* S is found before the **goto** *label* sentence, the function ρ gives the environment whose sentences we have to execute. The corresponding transition is:

$$\langle \textbf{goto}\ label, m, y, E, \rho, \gamma, s\rangle \rightarrow \langle \textbf{skip}, m, y, (\rho\ label), \rho, \gamma, s\rangle \qquad (\rho\ label) \neq nil$$

When we find a labeled statement in the top of the environment with a different label than the **goto** statement, we redefine the function ρ to answer the corresponding environment for the label.

$$\begin{array}{l}\langle \textbf{goto } label, m, y, (label' : S, n) : E, \rho, \gamma, s\rangle \rightarrow \\ \qquad \langle\ \textbf{goto } label, \\ \qquad\ m, \\ \qquad\ y, \\ \qquad\ E, \\ \qquad\ \lambda\ z.\ if\ (z = label')\ then\ (S, n) : E\ else\ (\rho\ z), \\ \qquad\ \gamma, \\ \qquad\ s\rangle\end{array}$$

When we find the statement $label : S$ on top of the stack our transition is

$$\begin{array}{l}\langle \textbf{goto } label, m, y, (label\ S, n) : E, \rho, \gamma, s\rangle \rightarrow \\ \qquad \langle\ \textbf{skip}, \\ \qquad\ n, \\ \qquad\ y, \\ \qquad\ (S, n) : E, \\ \qquad\ \lambda z.if\ (z = label)\ then\ (S, n) : E\ else\ (\rho\ z), \\ \qquad\ \gamma, \\ \qquad\ s\rangle\end{array}$$

If the sentence **goto** $label$ is inside an active **while**-loop and $label\ S$ is found below the **while**-loop we have to jump outside the **while**-loop. This is achieved with the transition

$$\begin{array}{l}\langle \textbf{goto } label, m, y, (\textbf{while } b\ \textbf{do } S, n) : E, \rho, \gamma, s\rangle \rightarrow \\ \qquad \langle\ \textbf{goto } label, \\ \qquad\ s(n), \\ \qquad\ y, \\ \qquad\ E, \\ \qquad\ \rho, \\ \qquad\ \lambda\ z.\ if\ (z = n)\ then\ false\ else\ (\gamma\ z), \\ \qquad\ s\rangle \qquad (\gamma\ n)\ = true \wedge s(n) \leq m\end{array}$$

Can be the case that *label* S appears after the **goto** *label* inside a non active **while**-loop, in this case we search *label* S in the body of the **while**-loop.

$$\begin{array}{l}\langle \textbf{goto}\ label, m, y, (\textbf{while}\ b\ \textbf{do}\ S, n) : E, \rho, \gamma, s\rangle \rightarrow \\ \qquad\qquad \langle\ \textbf{goto}\ label, \\ \qquad\qquad\ m, \\ \qquad\qquad\ y, \\ \qquad\qquad\ E, \\ \qquad\qquad\ h(S, 0, E, \rho), \\ \qquad\qquad\ \lambda\ z.\ if\ (z = n)\ then\ false\ else\ (\gamma\ z), \\ \qquad\qquad\ s\rangle\end{array}$$

label S can appear inside an **if** instruction:

$$\begin{array}{l}\langle \textbf{goto}\ label, m, y, (\textbf{if}\ b\ \textbf{then}\ S_1\ \textbf{else}\ S_2, n) : E, \rho, \gamma, s\rangle \rightarrow \\ \qquad\qquad \langle\ \textbf{goto}\ label, \\ \qquad\qquad\ m, \\ \qquad\qquad\ y, \\ \qquad\qquad\ E, \\ \qquad\qquad\ h(S_1, n, E, h(S_2, n, E, \rho), \\ \qquad\qquad\ \gamma, \\ \qquad\qquad\ s\rangle\end{array}$$

In any other case, we go ahead searching a statement of the form *label* S:

$$\langle \textbf{goto}\ label, m, y, (S, n) : E, \rho, \gamma, s\rangle \rightarrow \langle \textbf{goto}\ label, m, y, E, \rho, \gamma, s\rangle$$

To end, we define the semantic partial function
$S : \textbf{State} \rightarrow \textbf{State}$ by

$S(S)s = s'$ if and only if $initial(S)s = \langle \textbf{skip}, 0, 0, \alpha, \beta, s\rangle \rightarrow^* s'$

where
$\alpha = \lambda\ z.\ nil$ and $\beta = \lambda\ z.\ false$

5 Properties of the semantics

Definition 5.1. *Intermediate and final configurations*

Our transitions have the form

$\langle S, n, y, E, \rho, \gamma, s\rangle \to \delta$

where δ has the form $\langle S', n', y', E', \rho', \gamma', s'\rangle$ or the form s'. The first are called **intermediate configurations**, *and the last* **final configurations**.

We say $\langle S, n, y, E, \rho, \gamma, s\rangle$ is stuck *is there is no δ such that $\langle S, n, y, E, \rho, \gamma, s\rangle \to \delta$.*

Definition 5.2. *Derivation Sequence, termination, loops*

A derivation sequence is

1. *a finite sequence of configurations*

   $\delta_0, \delta_1, \ldots, \delta_k$

   where $\delta_i \to \delta_{i+1}$ for $0 \leq i < k$ and where δ_k is a final configuration or a stuck configuration.

2. *an infinite sequence*

   $\delta_0, \delta_1, \ldots$

   where $\delta_i \to \delta_{i+1}$ for $0 \leq i$.

We say that a statement S with counters n and y, environment E, functions α and β and state s, **terminates** *if and only if there is a finite derivation sequence starting in $\langle S, n, y, E, \alpha, \beta, s\rangle$ and* **loops** *if and only if there is a infinite derivation sequence starting in $\langle S, n, y, E, \alpha, \beta, s\rangle$.*

We say that a statement S from an state s, terminates if $initial(S)s$ terminates and loops if $initial(S)s$ loops.

Theorem 5.3. *There are no stuck configurations*

Proof. All the possible configurations are contemplated in our semantics. Whatever configuration we reach we have a transition. □

Theorem 5.4. *Determinism*
There is exactly one derivation sequence, starting in a configuration δ, i.e. if $\delta \to \delta'$ and $\delta \to \delta''$ then $\delta' = \delta''$.

Proof. There is only one possible transition for each possible pattern and condition. □

Corollary 5.5. *A statement S from an state s, terminates or loops.*

Proof. If S terminates, exists k and s' such that $initial(S)s \to^k s'$. If S from s loops, has an infinite derivation sequence. To both terminate and loop contradicts determinism. □

6 Related Work

The work in [2] presents a formal semantics based in *evolving algebras*. There treatment of **break**, **continue** and **goto** is to call a function *NextTask* that is primitive, static and belongs to one of the algebras, so is not specified its implementation.

The treatment in [5, 6] is rather different to our work, we base our definition in the index, while Norris defines special statement values (*BreakVal, ContVal, RetVal* and *StmtVal*) to whom occur transitions when a statement **break**, **continue**, **return** or an ordinary evaluation termination happens respectively.

He's semantics is big step and does not define the semantic of **goto**.

7 Conclusions

We have presented the semantics for **break**, **continue** and **goto** using an stack machine, with two counters and three functions.

It is known that we can express the same programs without using **break**, **continue** nor **goto**. In the present paper we have tried to show that the introduction of such constructions in giving program semantics is not so complicated nor so obscure as one could imagine before hand.

References

[1] H.M.Deitel & P.J.Deitel. *Como programar en C/C++*, Prentice Hall Hispanoamericana, S.A., 1995.

[2] Yuri Gurevich & James K. Huggins. *The Semantics of the C Programming Language*, EECS Departament, University of Michigan, Ann Arbor, MI 48109-2122, USA, February 1993.

[3] Matthew Hennesy. *The Semantics of Programming Languages, An Elementary Introduction using Structural Operational Semantics*, John Wiley & Sons, Chichester-New York-Brisbane-Toronto-Singapore, 1990.

[4] H.R.Nielson and F.Nielson. *Semantics with Applications, A Formal Introduction for Computer Science*, John Wiley & Sons.

[5] Michael Norrish. *An abstract dynamic semantics for C*, Computer Laboratory, University of Cambridge, 1997.

[6] Michael Norrish. *C formalised in HOL*, Computer Laboratory, University of Cambridge, 1998.

[7] Gordon Plotkin. *A structural Approach to operational semantics*, Technical Report DAIMI FN-19, Aarhus University,1981.

[8] Glynn Winskel. *The Formal Semantics of Programming Languages. An Introduction.*, The MIT Press, 1993.