Abstract

This work presents proof rules for termination and non-termination for a subset of a C-like language that consists of the statements: assignation, composition, if, skip and while and rules for termination of function calls.

1 Introduction

There are many books that treat the natural semantics and the structural operational semantics of C-like languages [1, 2, 3]. They usually define when a statement $S$ on a state $s$ terminates or loops, but does not give a way for proving the termination or non-termination of a program.

We define a formal system that allows to make these proofs.

An exception is [1] that gives rules for termination of statements except for while.

Work on termination of programs has been developed for logic programming, functional programming and concurrent programming but not for imperative programming.

Examples of the work developed are [4, 5, 10] for logic programming, [9, 6, 7] for functional programming and [8] for concurrent programming. More pointers to the literature can be find in these works.

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). We don’t present the semantics of expressions nor of statements because is widely presented in the literature. In section 3 we present the formal systems of our concern and examples of proofs in particular programs. In section 4 we extend the system to manage termination of functions. From the semantics of termination for functions one can infer a natural semantics for functions that was not presented before. In section 5 we present further work.

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
The meta-variables can be subscripted for example $a, a_1, a_2, \ldots$ all stand for arithmetic expressions.

**Definition 2.1. Abstract syntax for Arithmetic Expressions**

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

**Definition 2.2. Abstract syntax for Boolean Expressions**

$$b ::= \text{true} \mid \text{false} \mid a_1 = a_2 \mid a_1 < a_2 \mid \neg b \mid b_1 \lor b_2 \mid b_1 \land b_2$$

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

**Definition 2.3. Abstract syntax for Statements**

$$S ::= x := a \mid \text{skip} \mid S_1 ; S_2 \mid \text{if } b \text{ then } S_1 \text{ else } S_2 \mid \text{while } b \text{ do } S$$

### 3 Logics for termination and non-termination of programs

We present logics for *termination* and *non-termination* of programs together with examples of their application.

In what follows $A$ stands for the semantics function for arithmetic expressions and $B$ for the semantic function for boolean expressions.

#### 3.1 Logic for termination

Let $S$ stand for a program. We define the relationship $S$ in a state $s$ terminates on state $s'$. Table 1 presents the semantics of termination of statements.

**Theorem 3.1. Equivalence I**

An statement $S$ on a state $s$ terminates in an state $s'$ according to the natural semantics as defined in [2] if and only if $(S)s$ terminates on $s'$. **proof** straightforward.

**Theorem 3.2. Equivalence II**

An statement $S$ on a state $s$ terminates in an state $s'$ according to the structural operational semantics as defined in [2] if and only if $(S)s$ terminates on $s'$. **proof** straightforward.

**Example 3.3. The factorial program**

We prove the termination of the factorial program when is evaluated in the value 5.
(x := a)s terminates on s|x := A(a)S]

(skip)s terminates on s

(S1)s terminates on s' B(b)s = true
  (if b then S1 else S2)s terminates on s'

(S2)s terminates on s' B(b)s = false
  (if b then S1 else S2)s terminates on s'

(S1)s terminates on s'
  (S2)s' terminates on s''

(S1; S2)s terminates on s''

\exists j, 0 \leq j B(b)s_j = false
  \forall k, 0 \leq k < j B(b)s_k = true
  (S)_s_i terminates on s_{i+1}, 0 \leq i < j
  (while b do S)s terminates on s_j

Table 1: Semantics of Termination

y := 1
x := 5
while x > 0 do
  y := y * x
  x := x - 1

the initial state s is the empty function.
(y := 1)s terminates in s[y := 1].
(x := 5)s[y := 1] terminates in s_0 = s[y := 1][x := 5]
((y := y * x):(x := x - 1))s_i terminates on s_{i+1}
From s_0(x) = 5 and s_{i+1}(x) = s_i(x) - 1 follows s_5(x) = 0
and then B(x > 0)s_5 = false.

3.2 Logic for non-termination

Let S stand for a program. We define the relationship S in s state s does not
terminate. Table 2 presents the semantics of non-termination of statements.

Theorem 3.4. Equivalence I
An statement S on a state s loops according to the natural semantics as defined
in [2] if and only (S)s non-terminates. proof straightforward.

Theorem 3.5. Equivalence II
An statement S on a state s loops according to the structural operational seman-
tics as defined in [2] if and only if (S)s non-terminates. proof straightforward.

Example 3.6. A trivial looping program

while true do skip
$S_1$s non-terminates $B(b) = true$

(if $b$ then $S_1$ else $S_2$s non-terminates

$S_2$s non-terminates $B(b) = false$

(if $b$ then $S_1$ else $S_2$s non-terminates

$S_1$s non-terminates

$S_1;S_2$s non-terminates

$S_1$s terminates on $s'$ $S_2's'$ non-terminates

$S_1;S_2$s non-terminates

$∀k, 0 \leq k (S) s_k$ terminates on $s_{k+1} ∀j, 0 \leq j B(b)s_j = true$

(while $b$ do $S$s non-terminates

$∃i, 0 \leq i (S) s_i$ non-terminates $∀k, 0 \leq k \leq i B(b)s_k = true ∀k, 0 < i (S) s_k$ terminates on $s_{k+1}$

(while $b$ do $S$s non-terminates

Table 2: Semantics of Non-termination

$∀j, 0 \leq j (skip)s = s$ and $B(b)s_0 = true$
then $∀j, 0 \leq j B(b)s_j = true$ and the program does not terminate.

Example 3.7. A wrong factorial program

We prove the non-termination of a wrong factorial program whose loop-condition diverges.

$y := 1$
$x := 5$
while $x \leq 5$ do
$y := y * x$
$x := x - 1$

Let so be $s[y := 1][x := 5]$.
We prove by induction that $B(x \leq 5)s_i = true$ for all $0 \leq i$.

base) $B(x \leq 5)s_0 = true$
inductive step) assume $B(x \leq 5)s_i = true$ or what is equivalent $s_i(x) \leq 5$.
Since $s_{i+1}(x) = s_i(x) - 1$ then $s_{i+1}(x) < s_i(x)$
then $s_{i+1}(x) \leq 5$ and $B(x \leq 5)s_{i+1} = true$

Example 3.8. A third example

We prove the non-termination of a program that includes the wrong factorial program in an inner loop.

$x := 5$
while $x > 0$ do
$y := 1$
while $x \leq 5$ do
$y := y * x$
the initial state \( s \) is the empty function.
\((x := 5)s\) terminates in \( s_1 = s[x := 5] \).
\(B(x > 0)s_1 = true\)
If we call \( S \) the inner loop then \((S)s_1\) non-terminates.

4 Rules for functions

Any procedure can be written as a function that returns an arbitrary value, so we restrict ourselves to the treatment of functions.

We define a category of functions declarations called \( D \), a new syntactic category of statements \( S' \), a new syntactic category of expressions \( a' \) and some additional syntactic categories as \( \text{listofvars} \) for the parameters in the declaration of a function and \( \text{listofvalues} \) for the instantiation of the variables in a function call.

Definition 4.1. Abstract syntax for functions

\[
D ::= \text{function} \; \text{name} \;(\text{listofvars}) \; S' \\
S' ::= x ::= a \mid \text{skip} \mid S'_1; S'_2 \mid \text{if} \; b \; \text{then} \; S'_1 \; \text{else} \; S'_2 \mid \text{while} \; b \; \text{do} \; S' \mid \text{return}(a') \\
a' ::= \text{name}(\text{listofvalues}) \mid a'_1 + a'_2 \mid a'_1 * a'_2 \mid a'_1 a'_2 \\
\text{listofvars} ::= \text{var} \; x; \text{listofvars} \mid x; \text{listofvars} \mid \epsilon \\
\text{listofvalues} ::= a'; \text{listofvalues} \mid \epsilon
\]

\( \epsilon \) stands for the empty list and \( \text{name} \) for a string of letters and numbers.
The meta-variables in the primed categories are primed and can be subscripted for example \( a', a'_1, a'_2, \ldots \).
The arguments of the functions definition as well as the value returned are integers.

Definition 4.2. Environments for functions declaration

We define \( \text{Env} \) in \( \text{Names} \rightarrow S' \times \text{listofvars} \) and we store in it the declarations of functions. We define the semantic function \( \text{Decl} \) to process declarations of functions:

\[
\text{Decl}(\text{function name (listofvars)}\{S'\})\text{Env} = \\
\text{Env}[\text{name} := (S', \text{listofvars})]
\]

Definition 4.3. Semantics of return

We extend our definition of termination. In case a statement terminates we return also a value, that is the value of the arithmetic expression in the case of the statement return. For the definition of termination of the other statements (given before) we put by convention the value 0. We pass also as parameter the environment.

\[
A'(a')s \; e \text{ terminates on } n \\
\text{(return } a')s \; e \text{ terminates on } (s, n)
\]
4.1 Semantics of termination of function calls

The evaluation of arithmetic expressions can finish or not depending on the termination of function calls. We define the function $A'$ to give the semantics of the extended category of arithmetic expressions.

$$A' : a' \times \text{State} \times \text{Env} \rightarrow Z \times \text{State}$$

$(S')\text{changestate}(l, \text{listofvalues}, s)$ terminates on $(s', n)$ $\ e(name) = (S', l)$

\[
A'(\text{name}(\text{listofvalues}))s \ e = (n, \text{modifybyref}(l, s, s'))
\]

\[
A'(a'_1)s \ e = (n_1, s_1) \quad A'(a'_2)s_1 \ e = (n_2, s_2)
\]

\[
A'(a'_1 + a'_2)s \ e = (n_1 + n_2, s_2)
\]

$A'(a'_1)s \ e = (n_1, s_1) \quad A'(a'_2)s_1 \ e$ terminates on $(n_2, s_2)$

\[
A'(a'_1 + a'_2)s \ e = (n_1 + n_2, s_2)
\]

$A'(a'_1)s \ e = (n_1, s_1) \quad A'(a'_2)s_1 \ e$ terminates on $(n_2, s_2)$

\[
A'(a'_1 - a'_2)s \ e = (n_1 - n_2, s_2)
\]

$\text{changestate}(e, e, s) = s$

$\text{changestate}([\text{var } x : l_1], (a' : l_2), s) = \text{changestate}(l_1, l_2, s[x := A'(a')s])$

$\text{changestate}(x : l_1, (a : l_2), s) = \text{changestate}(l_1, l_2, s[x := A'(a')s])$

$\text{modifybyref}(e, s, s') = s$

$\text{modifybyref}([\text{var } x : l], s, s') = \text{modifybyref}(l, s[x := s'(x)], s')$

$\text{modifybyref}((x : l), s, s') = \text{modifybyref}(l, s, s')$

The function $\text{modifybyref}$ above changes the values of variables passed by reference (declared with $\text{var}$).

**Definition 4.4.** Statements with function calls and return

We have to define again termination for assignation because the expression can be a function call.

$$A'(a')s \ e = (n, s')$$

$$(x := a')s \ e = (0, s'[x := n])$$

In what refers to composition, since a statement return finishes the execution of the function, we redefine its semantics.

$(\text{return } a; S_2)s \ e$ terminates on $(s, A'(a)s e)$

\[
(S'_1)s \ e \text{ terminates on } (s_1, n_1) \quad (S'_2)s_1 \ e \text{ terminates on } (s_2, n_2)
\]

\[
(S'_1; S'_2)s \ e \text{ terminates on } (s_2, n_2)
\]

**Example 4.5.** The Fibonacci Function

**function** $\text{Fib}(n)$

{if $n = 0$ then return 1
  else if $n = 1$ then return 1
  else return($\text{Fib}(n - 1) + \text{Fib}(n - 2)$)}
The call $\text{Fib}(0)$ terminates if terminates the body of the function in an state $s$ in which $n$ is bound to 0. The body of the function terminates because terminates $(\text{return } 1)$ in $(s, 1)$. So $\text{Fib}(0)$ terminates in 1. Following the same reasoning, $\text{Fib}(1)$ terminates in 1.

Suppose we want to evaluate $\text{Fib}(2)$. The body of the function terminates if terminates return($\text{Fib}(1)+\text{Fib}(0)$). The evaluation of the expression ($\text{Fib}(1)+\text{Fib}(0)$) terminates on 2 by the corresponding rule of the function $A'$.

5 Further Work

We have presented rules for termination and non-termination for a subset of the C-language. Remains to study non-termination of recursive functions.

We find also interesting to study termination and non-termination of the statements break, continue and goto.

We can go further studying automatics termination analysis for imperative languages.

References


