Exit Statements are Executable Miracles

Wei Chen

In this paper, we present a simple wp semantics and a programming law for the exit statement.

Follow this and additional works at: https://openscholarship.wustl.edu/cse_research

Recommended Citation
https://openscholarship.wustl.edu/cse_research/671
Exit Statements are Executable Miracles

Wei Chen

WUCS-91-53

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
Saint Louis, MO 63130-4899
Exit Statements Are Executable Miracles

Wei Chen

Department of Computer Science
Washington University
Campus Box 1045
St. Louis, MO 63130

Abstract

In this paper, we present a simple wp semantics and a programming law for the exit statement.

Keywords: exit, miracle, wp, refinement.

1 Introduction

This paper has two principal goals. The first is to give a simple weakest precondition semantics for the exit statement. In order to define the semantics of exit, the Turing language [3] uses a more complicated form of the wp definition, which is a map over a triple of predicates. In contrast, our proposal maintains Dijkstra's original form and exploits a miraculous statement, one that can achieve impossibility, to represent exit.
The second purpose of this paper is to provide programming laws to develop programs that admit exit statements in the context of refinement calculus [1, 7, 6]. The Turing language supports a development methodology for exit only in the very restricted way that exit is the first or last statement of a loop. Our approach imposes no such restriction on the appearance of exit. And we have a dynamic programming law set. At various stages of program development, we stipulate new laws for later use. In this specific case, a new law will designate a miraculous statement to refine to an exit statement.

2 Refinement calculus and miraculous statements

The refinement calculus, as Carroll Morgan put in [6], is a notation and set of rules for deriving programs from their specifications. Our programming notation includes a specification statement to specify a programming task; thus there is no separate notation for specifications and the program derivations are carried out within a single framework.

Specifically, we extend Dijkstra's guarded command language with the following form of the specification statement [5]:

\[ x : [pre, post] \]

where \( x \) is a set of program variables, and \( pre \) and \( post \) are two predicates.

Its wp semantics is defined by

\[ wp(x : [pre, post], R) \equiv pre \land (\forall x : post : R) \]

Operationally, it specifies a programming task that when started in states satisfying \( pre \) terminates in states satisfying \( post \) by changing variables \( x \).
Obviously, this statement is not executable by a computer. Program development proceeds to eliminate all specification statements by applying programming laws. A collection of those laws can be found in [2].

Dijkstra’s law of the excluded miracle, i.e. \( wp(S, false) \neq false \) for any program \( S \), is not necessarily met by a specification statement. Consider \( x : [true, false] \). From the semantic definition of the specification statement, \( wp(x : [true, false], false) = true \), i.e. the statement guarantees to achieve everything, even impossibilities. We call any statement that does not meet the law of the excluded miracle miraculous.

In general, a miraculous statement cannot be further refined into an executable statement, so we should keep our program non-miraculous. However, admitting miraculous statements often simplifies the programming theory. In the following, we will see how we can use a miraculous statement to define the semantics of \( exit \), and we will also encounter a situation where a miraculous statement can be replaced by an executable statement.

3 The wp semantics of exit

First, we extend our programming notation. We assume that a \( do \) statement can be followed by a label \( (L) \), and that no two \( do \)'s can have the same labels. We will refer to an \( L \)-labeled \( do \) as \( do_L \). We also introduce \( exit \) in the syntax of \( exit(L) \). Operationally, it causes the program control to jump to the end of \( do_L \) when \( do_L \) encloses it.

We do not define \( exit \) independently. Since \( exit \) has a meaning only when it appears in some \( do_L \), we need only define \( do_L \) and deal with \( exit \) inside it. In calculating \( wp(do_L, R) \), suppose that the calculation eventually reduces
to \(wp(\text{exit}(L), Q)\) for some \(Q\). Since \(\text{exit}(L)\) changes program control to the end of \(do_L\) where \(R\) needs to hold, the weakest possible precondition to execute \(\text{exit}(L)\) is \(R\). However, \(wp\) requires that \(\text{exit}(L)\) establish \(Q\). Obviously, this is in general impossible (unless \(R \Rightarrow Q\)). Thus, we have to work out some miracle. In this case, we need to activate a miraculous statement whose weakest precondition is \(R\), i.e. :\([R, false]\) in our syntax.

Having observed this, we define the following exit rule

\[
wp(\text{do}_L(\text{exit}(L)), R) \equiv wp(\text{do}([R, false]), R)
\]

In other words, for a postcondition \(R\), \(wp\) of \(do_L\) is the same as unlabeled \(do\)'s after replacing all \(\text{exit}(L)\) with :\([R, false]\). As an example, we calculate

\[
wp(\text{do true} \rightarrow \text{exit}(L) \text{od}(L), R)
\]

\[
= \{ \text{exit rule} \}
\]

\[
wp(\text{do true} \rightarrow :[R, false] \text{od}, R)
\]

\[
= \{ \text{do rule} \}
\]

the strongest solution in terms of \(X\) in

\[
[X \equiv wp([R, false], X) \vee false]
\]

\[
= \{ \text{definition of the specification statement} \}
\]

the strongest solution in terms of \(X\) in

\[
[X \equiv R]
\]

\[
= \{ \text{calculus} \}
\]

\[
R
\]

4 Exit in refinement calculus

Now that we have a formal definition of \(\text{exit}\), the remaining question is how we can consciously introduce \(\text{exit}\) in program development. From our
semantics we see that we need to detect an adequate context where a certain type of specification statements can be replaced by an exit. Such a context appears when we introduce a do statement. The following law formalizes this idea.

**Law** (introduce $do_L$)

$$\neg B \wedge I \Rightarrow R$$

$$x:[I,R] \subseteq \text{do } B \rightarrow x:[vf = v \wedge I, I \wedge 0 \leq vf < v] \text{od}(L)$$

**Sublaw** (introduce $exit_L$)

$$y:[R,Q] \subseteq \text{exit}(L)$$

where $L$ is a fresh label, $vf$ and $v$ are an integer function and a fresh logical constant as usual.

We use sublaw to refer to a law which can be applied only within a block newly introduced by the application of the main law. In this case, the sublaw can be applied only to constructs within the $do_L$ introduced by the main law.

$S_0 \subseteq S_1$ indicates that $S_0$ can be replaced with $S_1$ (a formal definition of this refinement order $\subseteq$ can be found in [6]). Programming starts with a specification statement and proceeds by replacements under the refinement order until an executable program is reached. As an example, we derive

$$x:[x = 5, x = 5]$$

$$\subseteq \{ \text{the main law: } I, B, vf := x = 5, true, 0 \}$$

$$\text{do } true \rightarrow x:[v = 0 \wedge x = 5, x = 5 \wedge 0 < v] \text{od}(L)$$

$$\subseteq \{ \text{a known law "weaken pre": } x:[pre \wedge P, post] \subseteq x:[pre, post] \}$$
\( \text{do true} \rightarrow x : [x = 5, x = 5 \land 0 < v] \od(L) \)

\[ \subseteq \{ \text{the sublaw} \} \]

\( \text{do true} \rightarrow \text{exit } (L) \od(L) \)

That is, the final program terminates at \( x = 5 \), when started at \( x = 5 \).

5 Conclusion

We have formalized \textit{exit} within the \textit{wp} framework and provided programming laws to introduce it in program construction. A similar proof rule of \textit{exit} for partial correctness is given in [4]. Our use of the miraculous statement allows \textit{exit} to be easily adapted to the situation of total correctness. We anticipate the same techniques will be applicable to other forms of the \textit{goto} statement.

The \textit{exit} statement is rarely touched in formal program construction. One reason might be that there were no programming laws known about it. However, in some cases, using \textit{exit} can indeed lead to more straightforward programs. We hope that the techniques presented in this paper allow \textit{exit} to play a role in formal program development.

6 Acknowledgements

I thank Ken Cox, Jerome Plun and Bala Swaminathan for commenting on an earlier draft of this paper.

This work is prompted by a proof system for \textit{exit} reported by Ron Olsson and Daniel Huang [8], though Ron Olsson and I are still debating the validity of that system.
References


