Report Number: WUCS-91-53

1991-01-01

Exit Statements are Executable Miracles

Wei Chen

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

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

$$wp(x\!:\![pre, post], R) \;\widehat{=}\; pre \wedge (\forall\, x : post : R)$$

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 $\langle L \rangle$, 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$\langle L \rangle$. 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 $wp(\mathbf{exit}\langle L\rangle, Q)$ for some Q. Since $\mathbf{exit}\langle L\rangle$ changes program control to the end of do_L where R needs to hold, the weakest possible precondition to execute $\mathbf{exit}\langle L\rangle$ is R. However, wp requires that $\mathbf{exit}\langle L\rangle$ 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. : $[R, \mathit{false}]$ in our syntax. Having observed this, we define the following *exit* rule

$$wp(do_L(\mathbf{exit}\langle L\rangle), R) \;\; \widehat{=} \;\; wp(do(:[R, \mathit{false}]), R)$$

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

$wp(\mathbf{do}\, \mathit{true} \rightarrow \mathbf{exit}\,\langle L\rangle\, \mathbf{od}\langle L\rangle, R)$

= { *exit* rule }

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

= { *do* rule }

the strongest solution in terms of X in

$[X \equiv wp(: [R, \mathit{false}], X) \vee \mathit{false}]$

= { definition of the specification statement }

the strongest solution in terms of X in

$[X \equiv R]$

= { calculus }

R

4 *Exit* in refinement calculus

Now that we have a formal definition of *exit* , the remaining question is how we can consciously introduce *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)

$$\frac{\neg B \wedge I \;\Rightarrow\; R}{x:[I,R] \;\sqsubseteq\; \mathbf{do}\, B \rightarrow x:[vf = v \wedge I,\; I \wedge 0 \leq vf < v]\,\mathbf{od}\langle L\rangle}$$

Sublaw (introduce $exit_L$)

$y\!:\![R,Q] \;\sqsubseteq\; \mathbf{exit}\langle L\rangle$

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.

$S_0 \sqsubseteq S_1$ indicates that S_0 can be replaced with S_1 (a formal definition of this refinement order $\sqsubseteq$ 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

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

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

$\mathbf{do}\, true \rightarrow x\!:\![v = 0 \wedge x = 5, x = 5 \wedge 0 < v]\,\mathbf{od}\langle L\rangle$

$\sqsubseteq$ $\{$ a known law "weaken pre": $x\!:\![pre \wedge P, post] \sqsubseteq x\!:\![pre, post]$ $\}$

$$\mathbf{do}\, true \rightarrow\ x\!:\![x = 5, x = 5 \wedge 0 < v]\,\mathbf{od}\langle L\rangle$$

$$\sqsubseteq \quad \{\ \text{the sublaw}\ \}$$

$$\mathbf{do}\, true \rightarrow\ \mathbf{exit}\ \langle L\rangle\,\mathbf{od}\langle L\rangle$$

That is, the final program terminates at $x = 5$, when started at $x = 5$.

5 Conclusion

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

The *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 *exit* can indeed lead to more straightforward programs. We hope that the techniques presented in this paper allow *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 *exit* reported by Ron Olsson and Daniel Huang [8], though Ron Olsson and I are still debating the validity of that system.

References

[1] Back, R.J.R.: "A calculus of refinements for program derivations", *Acta Informatica* **25**, 593-624, 1988.

[2] Chen, W.: "Programming by transformation – theory and methods", D.Sc. Dissertation, Washington University (St. Louis). May 1991.

[3] Holt, R.C., Matthews, P.A., Rosselet, J.A. and Cordy, J.R.: *The Turing Programming Language: Design and Definition*, Prentice-Hall, Englewood Cliffs, 1988.

[4] London, R.L., Guttag, J.V., Horning, J.J., Lampson, B.W., Mitchell, J.G. and Popek, G.J.: "Proof rules for the programming language Euclid", *Acta Informatica* **10**, 1-26, 1978.

[5] Morgan, C.C.: "The specification statement", *ACM TOPLAS* **10**, 403-419, 1988.

[6] Morgan, C.C., Robinson, K. and Gardiner, P.: "On the refinement calculus", Technical Monograph PRG-70, Oxford University, 1988.

[7] Morris, J.M.: "Programs from specifications", in: *Formal Development of Programs and Proofs*, Addison-Wesley, Reading, MA, 81-115, 1990.

[8] Olsson, R.A. and Huang, D.T.: "Axiomatic semantics for *escape* statements", *IPL* **39**, 27-33, 1991.