A Closer Look To The Cycle Derivation

Rui Gustavo Crespo
Technical University of Lisbon,
Av. Rovisco Pais, 1049-001 Lisboa, Portugal
E-mail: rgc@digitais.ist.utl.pt

Abstract

The mathematical construction of programs, from a given formal specification of infinite systems, is the only way to prove the correctness of the solution. In this paper we analyse the identification of different cycle instructions, based on the axiomatic basis of programming. We review the reification of the specification into one unitary cycle, and further explore the reification into more tricky instructions, such as nested cycles, sequence of cycles, and combinations of cycles and tests. At the end we highlight how can we increase the reification process productivity, by a careful selection of the derived program.

1 Introduction

The axiomatic basis of programming, also known as *Hoare's logics* [9], is a well-known method to formally reify one given specification into a correct program. The specification paradigm adopted in this paper is based on the set-theory, similar to specification languages such as VDM [6] and Z [4]. The program specification is made of one pair of first-order logic formulae [1], the pre and post conditions (respectively Q and R).

Different methods have been studied in order to reify the specification into imperative language constructs, such as attribution, conditional execution and cycles [5]. Tools have also been developed for constructing proofs of program specifications expressed in Hoare's logics [8, 10].

Cycle statements are part in many algorithms, such as sorting and searching [11]. The derivation of programs made of cycle statements is also one of the most difficult tasks. The difficulty increases if, instead of one unitary cycle, the reification process produces instruction combinations, such as nested cycles and combinations of cycles and tests.

In this paper we analyse the reification of the specification into cycle combinations. We observe that, although the derived program and the number of proofs are larger, the reification process suffers minor changes. Furthermore, we highlight how careful identification of the derived program, such as multiple-guarded cycle instead conditional statements inside cycle, reduces the reification effort.

2 Wp transform

The correctness proof of the reification is based on the *weakest precondition transform*, *wp*, due to Dijkstra [3]. For a given program S and a postcondition R, $wp(\|S\|, R)$ returns the precondition that must hold, embracing the largest possible set of data, that guarantees R holds after the program execution.

For example, the weakest precondition of an assignment instruction is simply the required postcondition with the left-hand side variables replaced byt the right-hand side expressions.

$$wp(\|X := e\,\|, R) = R_e^X \qquad (1)$$

3 Cycle derivation

In this section we start with the reification of one specification into the simplest cycle, the unitary. We then extend the process of cycle reification into more complicated programs, enhancing the modifications necessary for the reification of the specification.

3.1 Unitary cycle

The derivation of one program made of one unitary cycle is widely available in the literature [5, 7].

In this process, the analyst starts to identify, from the postcondition, the formula valid inside the cycle (named invariant, P) and the function that controls the cycle progress (named variant function, t). The invariant is, simply, a weaker postcondition.

Secondly, the analyst must propose the cycle initialisation instruction S_{INIT}, that guarantees that the invariant holds inside the cycle. Then, the following equation must hold

$$Q \Rightarrow wp(\|S_{INIT}\|, P) \tag{2}$$

Thirdly, the analyst identifies the cycle guards $BB \equiv B_1 \vee \ldots B_n$ from the equation

$$(P \wedge \neg BB) \Rightarrow R_D \tag{3}$$

Fourtly, the analyst proposes one program for each guard B_i: these programs start with the application of the variant function t_+, which must be followed by another program segment S_i which restablishes the invariant. The following equation, therefore, must hold

$$(P \wedge B_i) \Rightarrow wp(\|t_+; S_i\|, P) \tag{4}$$

Altough the analist must propose the programs S_{INIT}, the guards and the program segment S_i, the correctness is verified only if equations (2), (3) and (4) hold. If, not, the error is highlighted and the analyst must change his proposal prior to the instalation of the computer application in its working environment.

Example 3.1 *Consider the specification of one program that sums one array of integers.*

$$Q : N \geq 1 \quad R : s = \sum_{i=1}^{N} A[i]$$

First, we need to identify the constructive postcondition R_D, replacing the limit (N) by one variable fixed to the limit.

$$R_D : s = \sum_{i=1}^{n} A[i] \wedge 1 \leq n \leq N \wedge n = N$$

The invariant P is, simply, weaker than R_D through the removal of the $n = N$ conjunction. This conjunction is named the equivalent condition.

$$P : s = \sum_{i=1}^{n} A[i] \wedge 1 \leq n \leq N$$

The variant is a monotonic decreasing function. Because n scans between 1 and N, it's easy to propose the variant function as $t = N - n$.

We may now start the second pass of the cycle reification. We propose $S_{INIT} \equiv n, s := 1, A[1]$ that initialises the free variables $\{s, n\}$, and check that equation (2) holds.

$$wp(\|n, s := 1, A[1]\|, s = \sum_{i=1}^{n} A[i] \wedge 1 \leq n \leq N) \equiv$$
$$A[1] = \sum_{i=1}^{1} A[i] \wedge 1 \leq 1 \leq N \equiv$$
$$true \wedge true \wedge 1 \leq N \equiv$$
$$1 \leq N$$

Obviously, $1 \leq N \Rightarrow 1 \leq N$ holds.

We start the third step with the identification of BB. $\neg BB$ is, simply, the equivalence condition. Therefore, $BB \equiv \neg n = N$.

We now may proceed to the final phase of the reification process. The statement $n := n+1$ applies the variant function. We propose the instruction $s := s + A[n+1]$, to restablish the invariant. To check our hypothesis, we must verify if equation (4) holds.

$$B_i \wedge P \equiv s = \sum_{i=1}^{n} A[i] \wedge 1 \leq n < N$$
$$wp(\|n, s := n+1, s + A[n+1]\|, P) \equiv$$
$$s + A[n+1] = \sum_{i=1}^{n+1} A[i] \wedge 1 \leq n+1 \leq N \equiv$$
$$s = \sum_{i=1} nA[i] \wedge 0 \leq n < N$$

$B_i \wedge P$ is stronger than $WP(\|t_+, S_i\|, P)$ and, therefore, we conclude the invariant holds inside the cycle.

The program that reifies the specification is then

```
n,s := 1,A[1];
do n ≠ N → n,s := n+1,s+A[n+1];
od
```

We should note that, when identifying the invariant P, the free variable used as a term in the removed conjunction, must also be a term outside the range formulae. If that is not the case, we must split the invariant range into two parts, as we shall see later in example 3.2.

In this section, we depicted the derivation of the unitary cycle, with attribution statements. However, many specifications are reified into other sorts of cycles. Therefore, we must adapt the method presented in this section.

3.2 Nested cycles

The reification of one specification into a program made of a nested cycle requires a change only in the last step of the unitary cycle derivation. If the proposed program for a guard B_i is a cycle, we must propose the specification of the inner cycle Q_i and R_i from the outer cycle invariant P_e. Then, we derive the inner cycle and, finally, we must check is P_e holds across the inner cycle.

The Q_i and R_i proposal is a creative step. This is not a problem, because the analyst must verify the validity of his proposal in the equation (4).

Example 3.2 *Consider the specification of one program that sums the rows of a two-dimension integer array into an one-dimension array.*

$Q_e : N \geq 1 \wedge M \geq 1$

$$R_e : (1 \leq j \leq M) \Rightarrow out[j] = \sum_{i=1}^{N} A[j,i]$$

The major difference to the specification in example 3.1, beyond the sum to an array element instead to one variable, the implication instead of a conjunction. This strategy allows the analyst to initialise free variables outside the range.

The outer cycle invariant P_e is

$$(1 \leq j \leq m \wedge 1 \leq m \leq M) \Rightarrow out[j] = \sum_{i=1}^{N} A[j,i]$$

Because the free variable m is not a term in a subformula outside the range, we must replace P_e by an equivalent formula by splitting the range

$$((1 \leq j < m \wedge 1 \leq m \leq M) \Rightarrow out[j] = \sum_{i=1}^{N} A[j,i]) \wedge$$
$$((j = m \wedge 1 \leq m \leq M) \Rightarrow out[m] = \sum_{i=1}^{N} A[m,i])$$

We propose the inner cycle post-condition be equal to the outer invariant. We also propose the pre-condition to be a conjunction between a formula holding true for subset of data covered by Q_e, and a formula holding true for subset of data covered by P_e: the subset of data covered by Q_e is the one focused by the inner cycle and the subset of data covered by P_e is the data still unprocessed by the inner cycle.

$$Q_i : N \geq 1 \wedge (1 \leq j \leq m \wedge 1 \leq m \leq M) \Rightarrow out[j] = \sum_{i=1}^{N} A[j,i]))$$

$R_i : P_e$

The inner cycle specification is similar to the example 3.1. Finally, we need to verify if equation (4) holds.

$$P_e \wedge B \equiv$$
$$m \neq M \wedge ((1 \leq j \leq m \wedge 1 \leq m \leq M) \Rightarrow out[j] = \sum_{i=1}^{N} A[j,i]) \equiv$$
$$(1 \leq j < m \wedge 1 \leq m < M) \Rightarrow out[j] = \sum_{i=1}^{N} A[j,i])$$

$$wp(\|t_+; S_i\|, P_e) \equiv$$
$$wp(\|t_+\|, wp(\|S_i\|, P_e)) \equiv$$
$$wp(\|t_+\|, (1 \leq j < m \wedge 1 \leq m \leq M) \Rightarrow out[j] = \sum_{i=1}^{N} A[j,i])) \equiv$$
$$(1 \leq j < m+1 \wedge 1 \leq m+1 \leq M) \Rightarrow out[j] = \sum_{i=1}^{N} A[j,i]) \equiv$$
$$(1 \leq j \leq m \wedge 1 \leq m < M) \Rightarrow out[j] = \sum_{i=1}^{N} A[j,i])$$

$P_e \wedge B$ and $wp(\|t_+; S_i\|, P_e)$ are equal, hence (4) holds. The program that reifies the specification is then

```
m := 0;
do m ≠ M →
   m := m+1;
   n,out[m] := 1,A[m,1];
   do n ≠ N →
      n,out[m] := n+1, out+A[m,n+1];
   od
od
```

3.3 Cycle and conditional statements

The reification of one specification into a program made of a conditional statement inside a cycle requires a large mathematical effort. We advocate the guards of the conditional statement should be transferred to the cycle guards BB, decreasing the reification effort.

Example 3.3 *Consider the specification of a program that counts the number of positive:* `N` *is the* *numerical quantifier, which returns the number of elements for which a formula is evaluated true.*

$Q : N \geq 1$

$R_D : 1 \leq n \leq N \wedge n = N \wedge$
$\quad p = \mathtt{N}j \bullet 1 \leq j \leq n \ \wedge A[j] > 0$

The reification of a conditional statement inside a cycle is presented in the literature [7]. The equivalence condition is $n = N$ and, therefore, $BB \equiv n \neq N$. In this paper, we replace BB by an equivalent formula

$BB \equiv (n \neq N \wedge A[n+1] > 0) \vee$
$\quad ((n \neq N \wedge A[n+1] \leq 0)$

The proof that, for $B_1 \equiv n \neq N \wedge A[n+1] > 0$, the equation (4) holds, is depicted next

$P \wedge B_1 \equiv$
$(0 \leq n \leq N \wedge p = \mathtt{N}j \bullet 1 \leq j \leq n \wedge A[j] > 0) \wedge$
$\quad n \neq N \wedge A[n+1] > 0 \equiv$
$(0 \leq n < N \wedge p+1 = \mathtt{N}j \bullet 1 \leq j \leq n+1 \wedge A[j] > 0)$

$wp(\|n, p := n+1, p+1\|, P) \equiv$
$(0 \leq n+1 \leq N \wedge p+1 = \mathtt{N}j \bullet$
$\quad 1 \leq j \leq n+1 \wedge A[j] > 0) \equiv$
$(0 \leq n < N \wedge p+1 = \mathtt{N}j \bullet 1 \leq j \leq n+1 \wedge A[j] > 0)$

$P \wedge B_1$ and $wp(\|n, p := n+1, p+1\|, P)$ are equal. For the second BB subformula, the proof is similar.

The program that reifies the specification is then

```
n,p := 0,0;
do n ≠ N ∧ A[n+1]>0 → n,p := n+1,p+1;
[ ] n ≠ N ∧ A[n+1]≤ 0 → n,p := n+1,p;
od
```

In example 3.3, the number of identified formulae, to check if equation (4) holds, is two, for each B_i guard. We also realised two inferences, one to replace BB and the other to validate the (4) equation.

If we choose to reify the specification into a program made of a conditional statement inside a cycle, the proof that equation (4) holds would require the identification of four formulae: one for each branch of the conditional statement, another for the combined application of the variant function with the conditional statement and, finally, the equation (4) itself. We would also have to realise two inferences, the adaptation of the ranges in both sides of the equation (4) and the validation of the (4) equation.

Taking in account only the number of identified formulae and the number of inferences, the reification into multiple-guarded cycle is lighter than into a conditional statement inside a cycle.

3.4 Instruction sequence

In section 3.3 we verified that it is possible to replace one conditional statement inside one unitary cycle, but a multiple-branch cycle.

In this section we explore another method of replacing the conditional statements inside one unitary cycle, the instruction sequence. In this method, we replace the postcondition by an equivalent formula of conjunctions, each one embracing a member of one data partition. Then, we reify the specifications for the unchanged precondition and every subformula and, finally, the final program is simply the sequence of subprograms. The validity of the method presented in this section is easily demonstrated using the distributivity and the null statement laws of the wp transform.

Example 3.4 *Consider the specification of a program that copies one array $Text[1..N]$, filled with characters from the beginning till an intermediate position L, into another array $out[1..N]$. In the copy process, one character C is inserted in the P position.*

$Q : N \geq 2 \wedge 1 \leq L < N \wedge 1 \leq P \leq L$

$R : (1 \leq i < P \Rightarrow out[i] = Text[i]) \wedge out[P] = C \wedge$
$\quad (P \leq i < L \Rightarrow out[i+1] = Text[i])$

Poscondition defines a partition of the output space

$R_1 : 1 \leq i < P \Rightarrow out[i] = Text[i]$

$R_2 : out[P] = C$

$R_3 : P \leq i < L \Rightarrow out[i+1] = Text[i]$

The program that reifies the specification can be made of a sequence of three programs, for the same Q and each R_i. R_1 and R_3 are reified into an unitary cycle. The program that reifies the specification could then be

```
/*R1 postcondition*/
l := 0;
do l ≠ P → l,out[l+1] := l+1,Text[l+1]; od
/*R2 postcondition*/
out[P]:=C;
/*R3 postcondition*/
l := P-1;
do l ≠ P → l,out[l+1] := l+1,Text[l]; od
```

The method of combining subprograms has been explored in non-isomorphic specification matching, to enable the development of large and complex systems from a repository of simple programs [2].

4 Conclusions

The identification of algorithms faces two major concerns, the correctness against the specification, and their complexity, in terms of execution time as a function of the data size [11].

We should add a third concern, the complexity of the correctness proof, in terms of the number of formulae we must identify and of the number of inferences we must realise. Because different algorithms may reify the same specification, if the execution time is not a major concern, analists should select the reification process that minimises the efforts of the correctness proof. The minimisation of the complexity of the correctness proof, not only eases the formal reification of a specification into a program, but also turns the proof more readable.

In this paper we highlighted several reification processes which, not only explore more cycle constructs, but also decrease the reification process efforts.

References

[1] Jon Barwise and John Etchemendy. *The Language of First-Order Logic*, volume 34 of *CSLI Lecture Notes*. Center for the Study of Language and Information, 1992.

[2] R.G. Crespo. Matching Single-Sort Algebraic Specifications for Software Reuse. *International Journal of Software Engineering and Knowledge Engineering*, 8(3):401–425, September 1998.

[3] E. W. Dijkstra. Nondeterminancy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, August 1975.

[4] Antoni Diller. *Z: An Introduction to Formal Methods*. John Wiley & Sons, 2 edition, 1994.

[5] Geoff Dromey. *Program Derivation, The Development of Programs From Specifications*. International Computer Science Series. Addison-Wesley, 1989.

[6] John Fitzgerald and Peter Gorm Larsen. *Modelling Systems, Prectical Tools and Techniques in Software Development*. Cambridge University Press, 1998.

[7] David Gries. *The Science of Programming*. Springer-Verlag, 1981.

[8] H. Peter Gumm. Generating algebraic laws from imperative programs. *Theoretical Computer Science*, 217:385–405, 1999.

[9] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580,583, October 1969.

[10] D. Kapur, X. Nie, and D.R. Musser. An overview of the Tecton proof system. *Theoretical Computer Science*, 133:307–339, 1994.

[11] Robert Sedgewick. *Algorithms in C*. Addison-Wesley, 1998.