Lightweight Formal Verification for Tail Recursive Loops

A. Ricardo Morales and J. Nelson Rushton
Department of Computer Science, Texas Tech University, Lubbock, Texas, U.S.A.

Abstract—*A formal method is given for generating a correctness argument from commented code for a tail recursive function. The generated argument is a set of propositions which, if true, guarantee the partial correctness of the code with respect to its documented specification (where by partial correctness, we mean that if the function returns anything at all then it returns the correct value). The intended use of the system is to teach students to write loop invariants.*

Keywords: Software Engineering, Program Verification, Functional Programming, Programming Pedagogy.

1. Introduction

This paper presents a formal method for generating a correctness argument from commented code for a tail recursive function. The method is formal in the sense that there is an algorithm for generating a set F of formulas from the commented code, such that if the propositions of F are true then the code is partially correct (that is, cannot return an incorrect value, though termination is not necessarily guaranteed). However, the checking of the propositions of F is left to intuition. Thus it might be said that the algorithm generates a proof, which is correct if the code is correctly written and commented — but that checking the proof is up to the programmer.

The *real* purpose of the method is to help teach students to comment code with preconditions, postconditions, and invariants. In order to teach invariants we need not only a precise definition of invariant, but also a precise definition of a "key" invariant that can be used in a correctness argument — for if we simply require students to write an invariant for a loop that fits the textbook definition, then $0 = 0$ will always do.

To make matters more challenging, we would like to give a criterion for key invariants that does not depend on the notion of a full formal argument. That is, we would like to define the class of invariants that *could* be used in a proof, independently of the notion of the proof itself. This allows invariants to be taught precisely, in contexts where there is not enough room in the syllabus to fully develop material on formal arguments.

For example, Code Sample 1 can be proven correct using the invariant $a = i!$ (in the absence of overflow errors, which will be addressed in Section 3). However, if we simply ask students to state *an* invariant of the tail recursive loop, then $0 = 0$ is is also a correct answer. What we seek is a precise definition that distinguishes key invariants from useless ones, without reference to their use in a full proof.

Code Sample 1:

```
; function fac
; precondition: n is a natural number
; postcondition: (fac n) = n!

(define (fac n) (facit 0 1 n))

(define (facit i a n) (cond
   [(= i n) a]
   [else (facit (add1 i) (* a (add1 i)) n)]))
```

The material presented here was used in a course on the theory of programming languages. By itself it took one day of class time. To understand it, students are required to have a basic familiarity with of formal logical propositions and functional programming, but *not* necessarily with formal inference rules or proofs. The most important learning outcome, in our view, is that students gain an intuition for writing invariants for tail recursive loops. The derivation of a correctness argument is simply a formalization of what *we* do intuitively in our heads, to convince ourselves that we have written an appropriate invariant.

2. Tail recursion

Code for a tail recursive function in Racket (a freely available dialect of Lisp) may be written as follows:

(`define` $(f\ a_1 \ldots a_n\ )($ $h\ S_1 \ldots S_m\ ))$
(`define`$(\ h\ x_1 \ldots x_m\ )$(`cond` $[\ G_1\ B_1\ ] \ldots [\ G_k\ B_k\ ]))$

where f, h, all a_i, and all x_i are symbols, and all S_i, G_i, and B_i are expressions. Each B_i must either be a base case which contains no recursive calls to h (or any function defined in terms of h), or a tail recursive case of the form $(h\ E_{i,1} \ldots E_{i,m}))$, where the $E_{i,j}$'s contain no calls to h (or any function which depends on h).

As a constraint on the use of this method, we assume that if f and h have any parameters in common, then these parameters are passed by h to itself unchanged in each recursive call. More precisely, if $x_i = a_j$ for some i and j, then for each recursive call $(h\ E_{k,1} \ldots E_{k,m})$ we have that $E_{k,j}$ is the symbol a_j.

3. Invariants

Good documentation for the tail recursive function shown in Code Sample 1 should also include an invariant. Loosely speaking, the invariant is a property of the arguments of the recursive helper h which has the following three properties:

1) it is true when h is first called,
2) it remains true for each recursive call to h, and
3) it guarantees a correct return value.

For example, our factorial function could be documented as follows:

$$;\texttt{invariant}:\texttt{a} = \texttt{i}!$$

In evaluating the expression (`fac 3`), for example, the following calls to `facit` will be generated, in order:

(`facit 0 1 3`)
(`facit 1 1 3`)
(`facit 2 2 3`)
(`facit 3 6 3`)

The invariant says that for each call (`facit` i a n) made during evaluation, we have $a = i!$. Note this is true in the trace above. Since execution stops only when $i = n$, this means we have $a = n!$ when execution halts. Since the expression returned is a, this implies that $n!$ is returned, as desired.

Next we will describe more precisely the properties which the invariant must have in relation to the pre- and post-conditions. The precondition is stated as a formula PRE whose free variables are among $a_1 \ldots a_n$. The postcondition is stated as a formula $POST$ whose free variables are also among $a_1 \ldots a_n$. The invariant is stated as a sentence INV whose free variables are among $x_1 \ldots x_m$.

Condition 1, that the invariant must hold on the first call to h, is written formally as

$$PRE \Rightarrow INV(S_1, \ldots, S_m) \quad (1)$$

where $INV(S_1, \ldots, S_m)$ is the sentence obtained from INV by substituting S_j for x_j, $1 \leq j \leq m$.

Condition 2, that the invariant remains true in each recursive call, corresponds, for each recursive case $[G_i\ (h\ E_{i,1} \ldots E_{i,m})]$ to a sentence of the form

$$INV \wedge \neg(G_1) \wedge \ldots \wedge \neg(G_{i-1}) \wedge G_i \Rightarrow INV(E_{i,1},\ \ldots,\ E_{i,m}) \quad (2)$$

where $INV(E_{i,1},\ \ldots,\ E_{i,m})$ is the sentence obtained from INV by substituting $E_{i,j}$ for x_j , $1 \leq j \leq m$.

Condition 3, that the invariant guarantees a correct return value corresponds, for each base case $[G_i\ B_i]$, to a sentence of the form

$$INV \wedge \neg(G_1) \wedge \ldots \wedge \neg(G_{i-1}) \wedge G_i \Rightarrow POST(B_i) \quad (3)$$

where $POST(B_i)$ is the sentence obtained from $POST$ by substituting B_i for $(f\ a_1 \ldots a_n)$.

These three formulas can be considered as premises for a valid argument that the defined function never returns an incorrect value. That is, if the formulas are true in the intended interpretation, then the function cannot returns an incorrect value.

In the case of our factorial function in Code Sample 1, conditions (1), (2), and (3) are instantiated as follows:

$$n \in \mathbb{N} \wedge n \geq 0 \quad \Rightarrow \quad 0 \neq 1 \quad (4)$$

$$a = i! \wedge \neg(i = n) \quad \Rightarrow \quad (*\ a\ (\texttt{add1}\ i)) = (\texttt{add1}\ i)! \quad (5)$$

$$a = i! \wedge i = n \quad \Rightarrow \quad a = n! \quad (6)$$

Unfortunately, statement (5) is not true. As a skeptical reader may have guessed before now, the supposed invariant fails to hold if execution of a recursive call results in an arithmetic overflow. One way to handle this is to write a new precondition and invariant that guarantee there is no arithmetic overflow. Racket has big integers built in, and it can store integers up to 1000! (and, in fact, much higher). Thus a fully documented factorial implementation can be written as follows:

Code Sample 2:

```
; function fac
; precondition: n in N  and 0<=n<=1000
; postcondition: (fac n) = n!

(define (fac n) (facit 0 1 n))

; invariant: 0<=i<=n<=1000 and a=i!

(define (facit i a n) (cond
    [(= i n) a]
    [else (facit (add1 i) (* a (add1 i)) n)]))
```

The corresponding (correct) premises are as follows:

$$n \in \mathbb{N} \wedge 0 \leq n \leq 1000 \quad \Rightarrow \quad 0! = 1 \quad (7)$$

$$0 \leq i \leq n \leq 1000 \wedge a = i! \wedge \neg(i = n) \Rightarrow 0 \leq (\texttt{add1}\ i) \leq n \leq 1000 \wedge (*\ a\ (\texttt{add1}\ i)) = (\texttt{add1}\ i)! \quad (8)$$

$$0 \leq i \leq n \leq 1000 \wedge a = i! \wedge i = n \quad \Rightarrow \quad a = n! \quad (9)$$

In a more general context, these three steps would be the base case, induction step, and final step of a a proof by induction that the function is partially correct (that is, cannot return an incorrect value). The induction proof can be automatically generated from these three key steps, and is correct if premises (1)-(3) are sound. This gives a method of proving that a tail recursive function never returns an incorrect value, which requires only three lines of documentation (at least two of which, and arguably all of which, ought to be written anyway), and a tool to generate the key proof steps so that they may be examined during a code review.

4. Another Example

Here is another simple example:

Code Sample 3:

```
;maximum
;precondition: L is a nonempty list of numbers
;postcondition: (maximum L)
;                  is the largest element of L

(define (maximum L) (maxit (car L) (cdr L)))

;invariant: MAXIMUM(L) = MAX(x,MAXIMUM(s))

(define (maxit x s) (cond
   [(empty? s) x]
   [(< x (car s)) (maxit  (car s) (cdr s))]
   [else (maxit x (cdr s))]))
```

For built-in functions, we will adopt the convention that an identifier in all caps denotes the mathematical function implemented by the corresponding operator in lower case letters. For user defined functions the convention is that the all-caps identifier denotes an ideal implementation satisfying $PRE \Rightarrow POST$. For example, here *MAXIMUM* is the mathematical function which is *supposed* to be implemented by the Racket function with the same name in lower case. This convention was found to be helpful because (1) the desired behavior will always need to be referred to in the argument for correctness, and (2) there is often no formal specification which is practically more clear than our pre-existing intuitive picture of the desired behavior. Some students seemed to think this was circular, but the following explanation helped: *MAXIMUM* is the function we want to write; $maximum$ is the function we actually wrote; and the question is whether they are the same. That question cannot be answered without talking about both.

The premises for the correctness argument for Code Sample 3 are as follows:

$$\begin{gathered}(L \text{ is a nonempty list of numbers}) \\ \Rightarrow \\ \textit{MAXIMUM}(L) = MAX((\texttt{car}\ L), \textit{MAXIMUM}(\texttt{cdr}\ L))\end{gathered} \tag{10}$$

$$\begin{gathered}\textit{MAXIMUM}(L) = MAX(x, \textit{MAXIMUM}(s)) \wedge \\ \neg(\texttt{empty?}\ s) \wedge x < (\texttt{car}\ s) \Rightarrow \\ \textit{MAXIMUM}(L) = MAX((\texttt{car}\ s), \textit{MAXIMUM}(\texttt{cdr}\ s))\end{gathered} \tag{11}$$

$$\begin{gathered}\textit{MAXIMUM}(L) = MAX(x, \textit{MAXIMUM}(s)) \wedge \\ \neg(\texttt{empty?}\ s) \wedge \neg\ x < (\texttt{car}\ s) \Rightarrow \\ \textit{MAXIMUM}(L) = MAX(x, \textit{MAXIMUM}(\texttt{cdr}\ s))\end{gathered} \tag{12}$$

$$\begin{gathered}\textit{MAXIMUM}(L) = MAX(x, \textit{MAXIMUM}(s)) \wedge \\ (\texttt{empty?}\ s) \Rightarrow \\ \textit{MAXIMUM}(L) = x\end{gathered} \tag{13}$$

One can check that these four statements are true, and infer from this the partial correctness of the code.

5. Conclusions and Future Work

This material was covered in one lecture of an hour and twenty minutes, in the junior level course *Concepts of Programming Languages* at Texas Tech, during Fall 2010. Four questions were given on the final exam which covered the material, requiring students to write tail recursive solutions for finding Fibonacci numbers and reversing a list, document their code with variants and invariants, and write formal premises corresponding to the preservation of truth of the invariant, and the guarantee of a correct return value. Out of 52 students who took the exam, 9 displayed mastery on these questions (as measured by receiving at least 90% credit on them) and 19 displayed either competency or mastery (by receiving at least 75% credit). This was a class where most students displayed competency or mastery on most material; and so these results seem to indicate a problem with either the material or the presentation. It should be noted that the course, of which this material was a major component, was well received in general, ranking higher than any other course in the department for Fall 2010 as measured by student responses to the question *was this course effective overall?*

We propose a two part hypothesis to explain the data:

1) Like all methods of rigorous or formal reasoning, the approach requires mathematical sophistication that must be built up over a period of years, and is generally not exercised by other classes in our curriculum.
2) The examples used to demonstrate the method were all of different sorts, requiring different mathematical concepts to invent an appropriate invariant.

Item (2) seemed like a good idea at the time. Indeed it is certainly beneficial to work several kinds of examples. However, having *no two examples alike* meant that students were loaded with novel concepts at every turn *in addition* to the target material.

We hypothesize that this can be improved by repeating several examples using the same design pattern (e.g., stepping through successive integers from low to high) before moving to another design pattern. Additionally, we hypothesize that the method itself can be streamlined in most cases by stating special case theorems that apply to common design patterns, such as recursively popping through a list or stepping through consecutive integers. This would be analogous to stating the Pythagorean Theorem and using it for the common special case of right triangles, instead of using the more general (and more complex) law of cosines. In the next iteration we plan to augment the theory with simplified methods for important special cases and demonstrate several uses of each.