Observations of Versatility of a Loop Analysis Method

Olfa Mraihi, Asma Louhichi, Lamia Labeled Jilani, 
Institut Superieur de Gestion Bardo 2000 Tunisia lamia.labeled@isg.rmu.tn

Ali Mili New Jersey Institute of Technology Newark, NJ 07102-1982 USA mili@cis.njit.edu

Abstract

This paper reports on ongoing work pertaining to the automated/systematic derivation of loop functions. The function of a loop captures, possibly in closed form, the effect of executing the loop on program variables. In this paper we illustrate it by means of a set of varied examples. We compare our results to those of other research efforts and conclude by outlining directions of further research.

Keywords

Function extraction, loop functions, invariant assertions, invariant relations, invariant functions, relational calculus, refinement calculus, computing loop behavior.

1 Deriving Loop Functions

Despite several decades of evolution, most programming languages in use today are Pascal-like languages, whose most important construct is the loop. Also, despite several decades of research, the analysis of programs for the purpose of understanding them, verifying their correctness or maintaining them remains a formidable challenge, for which there is little automated support. In this paper we discuss the design and preliminary implementation of an automated tool that analyzes the source code of a loop written in C, C++ or Java, and produces the function that this loop defines on its variables. Also, we illustrate the functionality of the proposed tool by applying it to three diverse examples.

In the following section we briefly present some mathematical background, which we use in section 3.1 to formulate a number of theorems that form the basis of our approach. In section 3.2 we briefly discuss the design and implementation of the tool, and in section 4 we show examples of application of the proposed method and tool. Finally, in section 5 we briefly summarize our results and assess them with respect to competing approaches and tools.

2 Mathematical Background

2.1 Elements of Relations

We consider a set $S$ defined by the values of some program variables, say $x$, $y$ and $z$; we typically refer to elements of $S$ by $s$, and we note that $s$ has the form $s = (x, y, z)$. We use the notation $x(s)$, $y(s)$, $z(s)$ to refer to the $x$-component, $y$-component and $z$-component of $s$. We use relations to represent program specifications and program functions. Constant relations on some set $S$ include the universal relation, denoted by $I$, the identity relation, denoted by $I$, and the empty relation, denoted by $\emptyset$. Given a predicate $t$, we denote by $I(t)$ the subset of the identity relation defined as follows: $I(t) = \{(s, s')| s' = s \land t(s)\}$.

Because relations are sets, we use the usual set theoretic operations between relations. Operations on relations also include the converse, denoted by $R$, and defined by $R = \{(s, s')| (s', s) \in R\}$. The product of relations $R$ and $R'$ is the relation denoted by $R \circ R'$ (or $RR'$) and defined by $R \circ R' = \{(s, s')| \exists t : (s, t) \in R \land (t, s') \in R'\}$. The domain of relation $R$ is defined as $\text{dom}(R) = \{s| s \in \text{dom}(R)\}$ and defined as $\text{dom}(R)$. The range of relation $R$ is denoted by $\text{rng}(R)$ and defined as $\text{dom}(R)$. We admit without proof that for a relation $R$, $RL = \{(s, s')| s \in \text{dom}(R)\}$ and $LR = \{(s, s')| s' \in \text{rng}(R)\}$. The nucleus of relation $R$ is the relation denoted by $\mu(R)$ and defined as $R\hat{\mu}$.

We say that $R$ is deterministic (or that it is a function) if and only if $RR \subseteq I$, and we say that $R$ is total if and only if $I \subseteq RR$, or equivalently, $RL = L$. A relation $R$ is said to be rectangular if and only if $R = RLR$. A relation $R$ is said to be reflexive if and only if $R \subseteq R$, transitive if and only if $RR \subseteq R$ and symmetric if and only if $R = R$.

2.2 Refinement Ordering

We define an ordering relation on relational specifications under the name refinement ordering:


Definition 1 A relation $R$ is said to refine a relation $R'$ if and only if
\[ RL \cap R'L \cap (R \cup R') = R'. \]

In set theoretic terms, this equation means that the domain of $R$ is a superset of (or equal to) the domain of $R'$, and that for elements in the domain of $R'$, the set of images by $R$ is a subset of (or equal to) the set of images by $R'$. This is similar to, but different from, refining a pre/postcondition specification by weakening its precondition and/or strengthening its postcondition [6, 10]. We abbreviate this property by $R \supseteq R'$ or $R' \subseteq R$. We admit that, modulo traditional definitions of total correctness [3, 6, 8], the following propositions hold.

- A program $P$ is correct with respect to a specification $R$ if and only if $[P] \supseteq R$, where $[P]$ is the function defined by $P$ (we may, by abuse of notation use $P$ to refer to $[P]$ when the context allows).
- $R \supseteq R'$ if and only if any program correct with respect to $R$ is correct with respect to $R'$.

Intuitively, $R$ refines $R'$ if and only if $R$ represents a stronger requirement than $R'$.

2.3 Refinement Lattice

We admit without proof that the refinement relation is a partial ordering. In [1] Boudriga et al. analyze the lattice properties of this ordering and find the following results:

- Any two relations $R$ and $R'$ have a greatest lower bound, which we refer to as the meet, denote by $\cap$, and define by:
  \[ R \cap R' = RL \cap R'L \cap (R \cup R'). \]

- Two relations $R$ and $R'$ have a least upper bound if and only if they satisfy the following condition:
  \[ RL \cap R'L = (R \cap R')L. \]

Under this condition, their least upper bound is referred to as the join, denoted by $\sqcup$, and defined by:
\[ R \sqcup R' = RL \cap R'L \sqcup RL \cap R \sqcup (R \cap R'). \]

- Two relations $R$ and $R'$ have a least upper bound if and only if they have an upper bound; this property holds in general for lattices, but because the refinement ordering is not a lattice (since the existence of the join is conditional), it bears checking for this ordering specifically.

- The lattice of refinement admits a universal lower bound, which is the empty relation.
- The lattice of refinement admits no universal upper bound.
- Maximal elements of this lattice are total deterministic relations.

See Figure 1. The outline of this figure shows the overall structure of the lattice of specifications.

3 Successive Approximations

3.1 Lower Bound Theorems

We consider a while statement of the form $w \Leftarrow w$ on some space $S$ and we are interested in deriving its function, which we denote by $W$. We assume that $w$ terminates normally for all states in $S$; this means that $W$ is total. The key of our approach is that we derive $W$ from statements of the form $W \sqcup T$, for various values of $T$; we refer to $T$ as a lower bound for $W$. If we can analyze the loop and derive lower bounds $T_1, T_2, \ldots, T_k$ of $W$, we write
\[ W \sqcup T_1 \wedge W \sqcup T_2 \wedge \ldots \wedge W \sqcup T_k, \]
from which we infer (by lattice theory)
\[ W \sqcup T_1 \sqcup T_2 \sqcup \ldots \sqcup T_k. \]

If $T_1 \sqcup T_2 \sqcup \ldots \sqcup T_k$ is total and deterministic, then (according to the lattice structure) it is maximal in the lattice, hence we infer
\[ W = T_1 \sqcup T_2 \sqcup \ldots \sqcup T_k. \]

If not, then the join of lower bounds forms a comprehensive lower bound of the loop function. In the remainder of this section we present a number of theorems that provide lower bounds for the loop function; proofs of these theorems are given in [9].

Theorem 1 We consider a while loop $w$ on space $S$, defined by $w = w \Leftarrow w$. If $R$ is a total transitive relation such that
\[ I(t) \circ [B] \subseteq R \]
and
\[ R \circ I(-t) \circ L = L, \]
then
\[ [w] \supseteq (R \cup I) \circ I(-t). \]
Theorem 2. We consider a while statement of the form
while \( t \) do \( B \) where the function of \( B \) can be written
as the union of two relations, \( P \) and \( Q \). If \( R \) and \( R' \) are
reflexive transitive relations such that
\[
P \subseteq R, \\
Q \circ R \subseteq R' \\
\text{and} \\
R \circ R' \circ I(-t) \circ L = L
\]
then
\[
W \supseteq R \circ R' \circ I(-t)
\]
i.e. \( T = R \circ R' \circ I(-t) \) is a lower bound for \( W \).

Theorem 3. Let \( w \) be a while loop defined by while \( t \) do \( B \). If \( t \neq \text{false} \) then
\[
[w] \supseteq T
\]
where \( T = I(t) \circ L \circ I(t) \circ [B] \circ I(-t) \cup I(-t) \).

Theorem 4. Given a while statement \( w \) of the form while \( t \) do \( B \) on space \( S \), such that \( w \) terminates for all initial
states in \( S \), and that \( t \neq \text{false} \). Then the following specifi-
cation is a lower bound of the function of the loop:
\[
T = (L \circ [B] \cup I) \circ I(-t).
\]

We distinguish between two classes of theorems, namely
constructive theorems (3 and 4) and creative theorems (1
and 2). Whereas the former produce constructive lower
bounds, the latter require creative steps in finding appropri-
ate relations \((R \text{ for Theorem 1 and } R \text{ and } R' \text{ for Theorem}
2)\).

3.2 A Tool for Loop Analysis

We use the theorems of the previous section as a basis in
the design of an automated tool that derives the function of a
while loop by analyzing its source code. The tool proceeds
in three steps:

- We map source code from the relevant languages to
  a unified language-independent notation. This allows
  us to keep the subsequent steps language-independent.
  For reasons that are beyond the scope of this paper,
  the target notation represents the loop body as a set
  of (possibly conditional) concurrent assignments; we
  refer to this notation as CCA.

- We analyze the internal notation for the purpose of ap-
  plying the lower bound theorems; the lower bounds we
  generate take the form of equations involving initial
  values and final values of the program variables. This
  step proceeds by matching CCA statements or com-
  binations of CCA statements against a library of code
  patterns and inferring appropriate lower bounds.
• Solve the equations generated above in the final values of the program variables, as a function of the initial values; this gives, in effect, the function of the loop. This step is carried out by a combination of a computer algebra system (Mathematica, Maple, Matlab) with a theorem prover.

4 Examples of Application

There is a wide range of methods and tools for the analysis of while loops, whether they involve the generation of loop invariants, loop functions, or recurrence relations. To characterize our approach, we present a set of loops that our tool can analyze.

4.1 Array Manipulations

We consider the following while loop on array variables $a$ and $b$, real variables $x$ and $y$, and index variables $i$ and $j$:

\[
\text{while } (i<N) \{x=x+a[i]; \ y=y+b[j]; \ i=i+1; \ j=j-1;\}
\]

We are interested in deriving the function of this loop; we are also interested to determine how this function is affected when the condition is changed to $(i<=N)$? to $(i!=N)$? to $(j>0)$? to $(j>=0)$? if we permute the index updates and the sum updates?

To answer these questions, we use the lower bounds provided by Theorems 1 and 3. Application of Theorem 1 requires a creative step, which we carry out by means of a pattern matching algorithm. Some of the patterns that we have used on these programs are given in table 2. Whenever we find a match of the program’s variables with the column labelled State Space and a match of the program’s CCA statements with the column labelled Code Pattern, we generate an instantiation of the relation given in the column labelled Lower Bound. These relations are represented directly in Mathematica, in preparation for the next step. Mathematica is used to solve these equations in the primed variables, using the unprimed variables as parameters. Table 3 summarizes the outcomes of various combinations of the proposed modifications, with the associated functions.

Each loop has been tested using randomly generated test data and using the associated function as an oracle; all of them have returned successful tests, as long as array indices are maintained within range.

4.2 A Fixpoint Iteration

We consider the following loop on variables $x$ and $y$ of type real:

\[
\text{while } (\text{abs}(x-y)>\epsilon) \{x=x; \ x=1+a/x;\}
\]

where abs is the absolute value, epsilon is a small real value, and $a$ is a real number. We are interested to determine the function of this loop; due to the roundoff errors of computers, we cannot derive its function precisely (i.e. we cannot tell what function it will compute on any particular computer), but derive an approximation of it. Theorem 1 yields no useful lower bound, as we cannot think of a reflexive transitive relation that is a superset of the loop body function. Theorem 3 yields no useful lower bound bound either, because knowing that at the previous iteration $|x-y|$ was greater than $\epsilon$ gives us no useful information on the function of the loop. Theorem 2 is not applicable to this loop, because the loop body is not a union. Hence we turn to theorem 4, which provides the following lower bound:

\[
T = \{ \text{Theorem 4} \}.
\]

\[
(L \circ [B] \cup I) \circ I(-t)
\]

\[
= \{ \text{Distributivity} \}
\]

\[
L \circ [B] \circ I(-t) \cup I(-t)
\]

\[
= \{ \text{Distributivity} \}
\]

\[
\{(s, s')|\exists s': y' = x \wedge x' = 1 + \frac{a}{y'} | x' - y' | \leq \epsilon \} \cup I(-t)
\]

\[
= \{ \text{Logic simplification} \}
\]

\[
\{(s, s')|x' = 1 + \frac{a}{y'} | x' - y' | \leq \epsilon \} \cup I(-t)
\]

Mathematica cannot solve the following equations in $x'$ and $y'$:

\[
x' = 1 + \frac{a}{y'} | x' - y' | \leq \epsilon.
\]

We simplify them by means of alternative interpretations of the clause $|x' - y'| \leq \epsilon$.

• $x' = y'$. Then, Mathematica solves the equations in $x'$ and $y'$ and produces the following outcome (two solutions):

\[
x' = \frac{1 - \sqrt{1 + 4a}}{2} \wedge y' = \frac{1 - \sqrt{1 + 4a}}{2}.
\]

\[
x' = \frac{1 + \sqrt{1 + 4a}}{2} \wedge y' = \frac{1 + \sqrt{1 + 4a}}{2}.
\]

• $x' = y' + \epsilon$. Then, Mathematica solves the equations in $x'$ and $y'$ and produces the following outcome (two solutions):

\[
x' = \frac{1 - \epsilon - \sqrt{1 + 4a - 2\epsilon + \epsilon^2}}{2} \wedge y' = \frac{1 - \epsilon - \sqrt{1 + 4a - 2\epsilon + \epsilon^2}}{2}.
\]
<table>
<thead>
<tr>
<th>ID</th>
<th>State Space</th>
<th>Code Pattern</th>
<th>Lower Bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>3R1</td>
<td>x: int</td>
<td>i=i+1,</td>
<td>( { (s, s')</td>
</tr>
<tr>
<td></td>
<td>a[N]: int</td>
<td>x = x+a[i]</td>
<td>( x + \sum_{k=1}^{N} a[k] = x' + \sum_{k=1}^{N} a'[k] )</td>
</tr>
<tr>
<td></td>
<td>i: int</td>
<td>a=a</td>
<td></td>
</tr>
<tr>
<td>3R2</td>
<td>x: int</td>
<td>i=i+1,</td>
<td>( { (s, s')</td>
</tr>
<tr>
<td></td>
<td>a[N]: int</td>
<td>x = x+a[i]</td>
<td>( x + \sum_{k=1}^{N} a[k] = x' + \sum_{k=1}^{N} a'[k] )</td>
</tr>
<tr>
<td></td>
<td>i: int</td>
<td>a=a</td>
<td></td>
</tr>
</tbody>
</table>

Figure 2. Array Loops and their Functions

Figure 3. Array Sum Patterns
\[ x' = \frac{1 - \epsilon + \sqrt{1 + 4a - 2\epsilon + \epsilon^2}}{2} \]
\[ y' = \frac{1 - \epsilon + \sqrt{1 + 4a - 2\epsilon + \epsilon^2}}{2}. \]

Both of these are viable approximations of the function computed by the loop. We have tested this loop with an oracle derived from these functions by replacing the equality with an approximation; for example, for the case \( x' = y' \), we write the following oracle

```java
boolean oracle (real x, y, xP, yP)
{ return (abs(x'-((1-sqrt(1+4a))/2) < epsilon)
& & abs(y'-(1-sqrt(1+4a))/2) < epsilon)
| | (abs(x'-(1+sqrt(1+4a))/2) < epsilon)
& & abs(y'-(1+sqrt(1+4a))/2) < epsilon) ; }
```

where \( \varepsilon = 10^{-6} \). We produce test data by varying \( \varepsilon \) and by varying \( a \); all tests for which the loop converges are successful with respect to the selected oracle.

If, for the sake of argument, we augment the loop with an integer variable \( i \), that we decrement at each iteration, then we get:

```java
while (abs(x-y) > epsilon)
{ y=x; x=1+a/x; i=i-1; }
```

Then Theorem 1 produces another lower bound, namely

\[ \{(s', s') | f^i(x) = f^i(x') \land i \geq i' \land |x' - y'| \leq \epsilon \}, \]

where \( f(x) = 1 + \frac{x}{2} \). We rewrite this relation as:

\[ \{(s', s') | x = f^{-i'}(x') \land i \geq i' \land |x' - y'| \leq \epsilon \}. \]

Given that we already know \( x' \) (equations above), this can be used to characterize \( (i - i') \), hence allows us to derive \( i' \) from \( i \).

### 4.3 An Integer Logarithm

We consider the following loop on integer variables \( x \) and \( y \):

```java
while (x>1)
{ if (x%4 ==0) {x = x div 4; y = y+2;}
 else {x = x div 2; y = y+1;}}
```

We find that the function of the loop body can be written as the union of two terms,

\[ P = \{(s', s') | x \mod 4 = 0 \land x' = x/4 \land y' = y + 2 \}, \]
\[ Q = \{(s', s') | x \mod 4 \neq 0 \land x' = x/2 \land y' = y + 1 \}. \]

We apply theorem 2 to this loop, whose first step involves finding a reflexive transitive superset of \( P \). Using pattern matching artifacts of the type we use for Theorem 1, we find the following relation that is a reflexive transitive superset of \( P \):

\[ R = \{(s', s') | y + \log(x) = y' + \log(x') \land y \mod 2 = y' \mod 2 \}. \]

According to Theorem 2, we must now find a relation \( R' \) that is a reflexive transitive superset of \( QR \); to this effect, we compute \( QR \):

\[
QR = \begin{cases}
\text{Substitutions} & \{(s', s') | x \mod 4 \neq 0 \land x' = x/2 \land y' = y + 1 \} \land \{(s', s') | y + \log(x) = y' + \log(x') \land y \mod 2 = y' \mod 2 \}.
\end{cases}
\]

We let this be relation \( R'' \), and we apply Theorem 2; this yields (after simplification) the following lower bound:

\[ T_1 = \{(s', s') | y + \log(x) = y' + \log(x') \land -x' > 1 \}. \]

We apply Theorem 3 to this loop, which yields (as the reader can easily check):

\[ T_2 = \{(s', s') | x' = 1 \}. \]

The join of these lower bounds produces a total deterministic relation, which is the function of the loop:

\[ W = \{(s', s') | x' = 1 \land y' = y + \log(x) \}. \]

### 5 Conclusion: Summary, Assessment, Prospects

#### 5.1 Summary

In this paper we discuss a method for deriving the function of a loop by successive approximations in a lattice. We
have presented four theorems that form the basis of our approach then we have outlined the algorithm that we are using to derive loop functions by application of the proposed theorems. Then we have illustrated the application of our algorithm on three different examples: an array manipulation example, which illustrates how small changes in the text of a program can affect the function of a program; a fixpoint iteration, which shows how we can derive the function of a loop in the absence of an inductive argument (the loop processes real numbers, and has no inductive variable, such as a loop counter); a program whose loop body has a non-trivial (non-sequential) structure.

5.2 Assessment and Comparison

The analysis of while loops for the purpose of deriving loop invariants, loop functions, or recurrence relations has experienced a recent upsurge, after a relative loss of interest in the late eighties and throughout the nineties. There are far too many references dealing with these topics for us to do them all justice; hence we will limit ourselves to citing those that are closest to our work, be it in terms of goals or in terms of methods.

We cite three lines of research that are closely related to our work: research on deriving loop functions, with which it shares a common goal; research on deriving loop invariants, with which it shares common analytical methods; and research on program slicing, with which it shares common divide-and-conquer approaches. We discuss these in turn, below.

The closest work we have found to our effort, in terms of goal (generating loop functions) and means (using Mills-like functional/relational logic) is work by Dunlop and Basili [4]. In this work, Dunlop and Basili discuss a syntactic method that derives the function of a loop by attempting to generalize from known formulas that capture the behaviors of the loop under special conditions. Dunlop and Basili’s approach is very syntactic, and uses a very small set of rules, that has limited scope of application. Another more recent, closely related work, is that of Carette and Janicki [2]; in this work, Carette and Janicki capture the function of while loops by recurrence relations, which they submit to a Computer Algebra System (such as Mathematica) to solve. The scope of our algorithm is broader than that of Carette and Janicki, because theirs is limited to numeric algorithms, and can only be applied when the function of the loop can be captured by a recurrence relation. The example of the fixpoint iteration, for example, does not fall in their scope. Interestingly, the lower bounds that we derive in Theorem 1 are very similar to recurrence relations from which the inductive variable has been cancelled out.

In [5] Ernst et al. discuss a system for dynamic detection of likely invariants; this system, called Daikon, runs candidate programs and observes their behaviors at user-selected points, and reports properties that were true over the observed executions, using machine learning techniques. Because these are empirical observations, the system produces probabilistic claims of invariance. Notwithstanding the fact that Daikon results from a development effort that is perhaps several orders of magnitude bigger than our preliminary/ rudimentary prototype, the Daikon approach differs from our in many ways: Daikon produces invariant assertions with respect to a pre-specification post-specification, whereas we derive the function of the loop; Daikon is deployed on executable code, hence requires that all constants be instantiated, whereas we analyze source code, hence can accommodate all kinds of parameters; Daikon produces likely invariants, whereas we produce provably correct functions or lower bounds. The great advantage of the Daikon approach is that it handles executable code, hence can be applied to any iterative control structure, whereas we can only handle previously modeled structures.

In [7] Hu et al present a technique for slicing while loops while attempting to minimize slice sizes. The technique is based on identifying the induction variable of the loop, and applying semantics-preserving transformations that represent the effect of the loop by an if-then-else statement. Our work differs from that of Hu et al. in many ways, including:

- First, we do not need to identify an inductive variable (we can think of cases where no such a variable can be defined, let alone identified); by finding reflexive transitive supersets of the loop body, we in fact do away with the inductive argument altogether.
- Second, our lower bounds can be arbitrarily partial, as they are not driven by the syntactic structure of the loop (while slicing techniques slice the program, our divide-and-conquer techniques slices the program’s function).
- Third, the relation of our lower bound to the function of the loop is well defined (refinement), as is the rule for composing lower bounds (join); by contrast, we could not find provable claims that elucidate the relation between the function of the slices that Hu et al extract, and the function of the whole loop.

5.3 Prospects of Further research

Practical extensions of our work include, obviously, expanding its scope by finding and deploying more lower bounds, to deal with more general data structures and control structures. Also, we envision to streamline the generation of lower bounds, so as to control its time performance and reduce the redundancy of the equations it generates. Another focus of our future research is the resolution of the
equations that stem from lower bounds; we are considering
a combination of a computer algebra system (e.g. Mathe-
matica) and a theorem prover (e.g. Otter).

The most critical theoretical extension that we must en-
vision is to generalize the algorithm to deal with arbitrary
(or more general) structures of the loop body. Whereas the-
orem 2 offers a basis for handling if-then-else statements,
possibly nested, its implementation is likely to pose a num-mer of practical problems, and to complicate the function
extraction algorithm rather significantly.

These matters are currently under investigation.

References

[1] N. Boudriga, F. Elloumi, and A. Mili. The lattice of speci-
1976.
functions. IEEE Transactions on Software Engineering,
 C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon sys-
tem for dynamic detection of likely invariants. Science of
1981.
squashing transformations for amorphous slicing. In Pro-
ceedings, 11th Working Conference on Reverse Engineering.
graw Hill, 1974.
soning about loop functions. Technical report, New Jersey
Institute of Technology, http:
Series in Computer Sciences. Prentice Hall, London, UK,
1998.