Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 243 (2009) 139–155

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Harnessing a Refinement Theory to Compute Loop Functions

Ali Mili [a,1] Rahma Ben Ayed [b,2] Shir Aharon [a,3]
Chaitanya Nadkarni [a,4]

[a] *College of Computing Science*
New Jersey Institute of Technology
Newark NJ 07102, USA

[b] *SysCom Ecole Nationale d'Ingénieurs de Tunis*
University of Tunis El Manar
Tunis, Tunisia

Abstract

We consider a while loop on some space S and we are interested in deriving the function that this loop defines between its initial states and its final states (when it terminates). Such a capability is useful in a wide range of applications, including reverse engineering, software maintenance, program comprehension, and program verification. In the absence of a general theoretical solution to the problem of deriving the function of a loop, we explore engineering solutions. In this paper we use a relational refinement calculus to approach this complex problem in a systematic manner. Our approach has many drawbacks, some surmountable and some not (being inherent to the approach); nevertheless, it offers a way to automatically derive the function of loops or an approximation thereof, under some conditions.

Keywords: Reverse engineering; software maintenance; program comprehension; while loops; program semantics; program correctness; refinement calculi; software tools.

1 Introduction

As software is used in increasingly critical applications, it is getting increasingly important to ensure its correctness, and to analyze/ understand its function. Simultaneously, as software grows increasingly large and complex, it is getting more and more difficult and costly to do so to an adequate level of confidence. Furthermore, recent software development paradigms (software reuse, product line engineering, COTS based software development, outsourcing, etc) are heavily dependent on third

[1] Email: mili@cis.njit.edu

[2] Email: rahma.benayed@enit.rnu.tn

[3] Email: ShirAharon09@comcast.net

[4] Email: cgn4@njit.edu

doi:10.1016/j.entcs.2009.07.010

party software products, whose quality cannot be ascertained by process controls (process standards, process maturity levels, etc); this places the burden of quality assurance on analyzing the resulting product. The convergence of these three trends places a great premium on automated tools that allow us to analyze the function of software components and software systems to an arbitrary level of thoroughness and precision.

Deriving or approximating (characterizing) the function of a software system involves reasoning at many different levels of the software hierarchy, and modeling many aspects of interaction between the components of a complex system. At the lowest level, the source code level, one of the most challenging tasks is the derivation or the approximation of loop functions. In this paper, we present some mathematical results that pertain to the approximation of the function of a loop in terms of inequalities in a refinement calculus. Then we use these results to design and (very) partially implement an algorithm that derives the function of a while loop from a static analysis of its source code. We adopt the following premises as guidelines in approaching this problem.

- *Closed Form Function.* Central to the derivation of a loop function is the discovery (reverse engineering) of the inductive argument that gave rise to the loop in the first place; giving the loop function by a recursive formula merely replaces an inductive argument by another. We resolve to derive the function of the loop in closed form, by describing how the execution of the loop affects all relevant variables of the program.
- *Stepwise Derivation.* We derive the function of a loop in a stepwise manner, by analyzing arbitrarily small parts of it, from which we infer arbitrarily small functional details about it. This allows us to handle arbitrarily large loops with relatively little complexity overhead.
- *Partial Analysis.* Even when we cannot derive the function of a loop in all its detail, we can still make statements about its function, on the basis of whatever parts of the loop fall within the scope of application of the algorithm.

These premises will be elucidated in the sequel, primarily in section 5. In the next section we showcase the current capability of our algorithm by means of a sample program, whose function we compute using our algorithm. Then, in section 3 we present the broad structure of our algorithm, and discuss its current status of development. In section 4 we briefly present the mathematical foundations of the algorithm, and use these to present the detailed structure of the algorithm, in section 5. In section 6 we assess the proposed algorithm, outline its future evolution in light of this assessment, and briefly discuss related work.

2 Brief Illustration

The purpose of this section is two-fold: first to showcase the current capability of our algorithm; second, to convey to the reader what we mean by *deriving a loop function.* We consider the C++ program given in figure 1, and we are interested to derive the function of its loop. This program handles integer variables, and also includes arrays, lists and (symbolic) function calls. For the sake of simplicity, we assume that constants a, and b are different from 0, and that constant d is different from 0 and 1 (without these hypotheses, the expression of the function would be very complex). Also, we assume that variable i is non-negative, and that it causes no failure of this loop (indices i and j remain within range of their arrays, and the length of list l is greater than or equal to i) [5].

The function of this loop is given in figure 2 (where list concatenation is represented by a dot). It includes two terms: the trivial term where $i = 0$ and all variables are preserved; the non-trivial term where $i \neq 0$ and program variables are altered. This figure gives the final values (primed) of the program variables as a function of the initial values (unprimed). For the sake of comparison, we submitted the same program to Daikon [10], which generates loop invariants by applying machine learning techniques to the execution trace. Because it operates on execution traces (rather than on source code), Daikon requires that we fix all the constants (a significant loss of generality, since then it makes a statement not about a broad family of programs, but rather about a single program). Daikon did find some of the clauses of the function given in Figure 2, duly specialized to the constant values.

3 Broad System Structure

To derive the function of a loop written in a given programming language, we proceed in three steps.

(i) *Map the loop from its source programming language notation to a predefined language-independent internal notation.* The internal notation is defined in such a way as to support the divide and conquer approach that we advocate. We make it language independent so as to support a wide range of programming languages with minimal overhead.

(ii) *We analyze the loop written in the internal notation to derive equations between the initial (unprimed) variables and the final (primed) variables.* This step is the core of our algorithm. We analyze small parts of the loop at a time with a view to answering the question: What equations hold between the initial values and the final values of the loop.

(iii) *We submit the equations derived in the previous step to a system for solving symbolic equations.* We obtain the function of the loop by solving the equations in the primed variables, using the unprimed variables as parameters. For now we are using *Mathematica* (©Wolfram Research), but we are also exploring other systems as well.

The first step is currently carried out by hand, but can easily be automated using compiler generation technology. The third step is fairly trivial, since the equations generated by the second step are written directly in Mathematica notation. Never-

[5] The current algorithm can automatically generate some of these conditions; we are currently exploring means to automatically generate all of them.

```
 #include <iostream>
 #include <cmath>
 #include <math.h>
 #include <list>
 using namespace std;
 const int a= , b= , c= , d= , e= ;
 const int N= ;
 typedef list <int>
 listtype;
 listtype l;
 listtype m;
 int q, qc;
 int x, y, z, t, i, j, v, w, SA, Sn;
 int A[N], B[N];
 void loop ();
 int f (int x);
 int main()
    {loop();}
 void loop  ()
    {
    while (i!= 0)
        {y = y+b;
        v = v+a*t;
        w = w+e*y-b*e;
        x = x+a;
        t = t*d;
        sA = sA + A[i];
        sB = sB + B[j];
        i = i-1;
        z = z+c*x-a*c;
        j = j+1;
        m.push_back(l.front());
        l.pop_front();
        q = f(q);
        qc = qc + q;}
   }
int f (int x)
   {return (//some function of
            x);}
```

Fig. 1. Sample C++ Program

$$\left\{\left(\begin{array}{cccc} x & y & z & v \\ w & t & i & j \\ sA & sB & A & B \\ l & m & q & qc \end{array}, \begin{array}{cccc} x' & y' & z' & v' \\ w' & t' & i' & j' \\ sA' & sB' & A' & B' \\ l' & m' & q' & qc' \end{array}\right) \middle| \begin{array}{c} d \neq 1 \wedge abdei \neq 0 \wedge i \leq len(l) \wedge i' = 0 \wedge \\ v' = \frac{(atd^i + vd - at - v)}{(d-1)} \wedge t' = d^i t \wedge q' = f^i(q) \wedge \\ w' = \frac{bei^2 - bei + 2eyi + 2w}{2} \wedge x' = x + ai \wedge y' = y + bi \wedge \\ z' = \frac{aci^2 - aci + ecxi + 2z}{2} \wedge qc' = qc + \sum_{k=1}^{i} f^k(q) \wedge \\ sA' = sA + \sum_{k=1}^{i} A[k] \wedge sB' = sB + \sum_{k=j}^{j+i-1} B[k] \wedge \\ j' = j + i \wedge l' = Rst^i(l) \wedge m'.Rst^i(l) = m.l \end{array} \right\}$$

$$\bigcup$$

$$\left\{\left(\begin{array}{cccc} x & y & z & v \\ w & t & i & j \\ sA & sB & A & B \\ l & m & q & qc \end{array}, \begin{array}{cccc} x' & y' & z' & v' \\ w' & t' & i' & j' \\ sA' & sB' & A' & B' \\ l' & m' & q' & qc' \end{array}\right) \middle| \begin{array}{c} i = 0 \wedge abd^2e \neq abde \wedge m' = m \wedge \\ x' = x \wedge y' = y \wedge z' = z \wedge t' = t \wedge v' = v \wedge \\ w' = w \wedge i' = 0 j' = j \wedge sA' = sA \wedge sB' = sB \wedge \\ l' = l \wedge q' = q \wedge qc' = qc \wedge A' = A \wedge B' = B \end{array} \right\}.$$

Fig. 2. Function of the Sample C++ Program

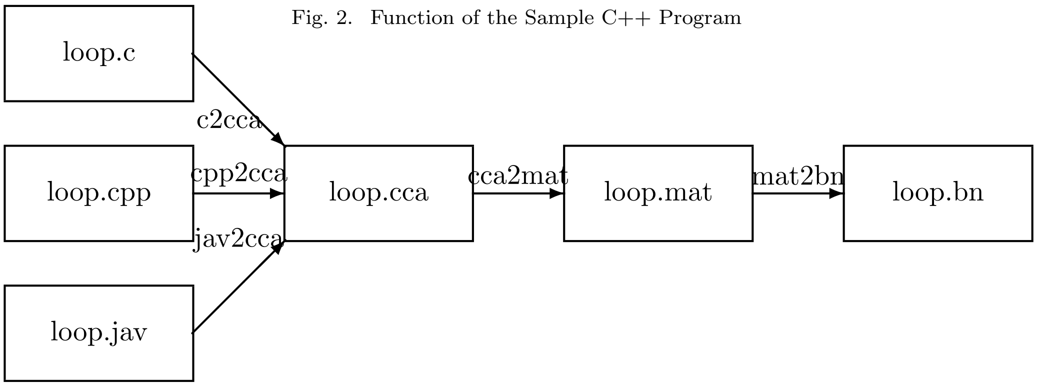

Fig. 3. Broad Architecture of the Tool

theless, this step is currently the bottleneck of our capability, in the sense that it determines what aspects of a program we can or cannot handle. The second step is the focus of our subsequent discussion.

4 Mathematical Foundations

4.1 Relational Mathematics

We represent the functional specification of programs by relations; without loss of generality, we consider homogeneous relations, and we denote by S the space on which relations are defined. A relation R on set S is a subset of the Cartesian product $S \times S$, hence it is natural to represent general relations as $R = \{(s, s')|\ p(s, s')\}$, for some predicate $p(s, s')$. Typically, set S is defined by some variables, say x, y, z; whence an element s of S has the structure $s = \langle x, y, z\rangle$. We use the notation $x(s)$, $y(s)$, $z(s)$ (resp. $x(s')$, $y(s')$, $z(s')$) to refer to the x-component, y-component and z-component of s (res. s'). We may, for the sake of brevity, write x for $x(s)$ and x' for $x(s')$.

Constant relations include the *universal* relation, denoted by L, the *identity* relation, denoted by I, and the *empty* relation, denoted by ϕ. Given a predicate t, we denote by $I(t)$ the subset of the identity relation defined as follows: $I(t) = \{(s, s')|\ s' = s \wedge t(s)\}$. Because relations are sets, we use the usual set theoretic operations between relations. Operations on relations also include the *converse*, denoted by $\widehat{R}$ and defined by $\widehat{R} = \{(s, s')|(s', s) \in R\}$. The *product* of relations R and R' is the relation denoted by $R \circ R'$ (or RR') and defined by $R \circ R' = \{(s, s')|\exists t : (s, t) \in R \wedge (t, s') \in R'\}$. The *prerestriction* (resp.*post-restriction*) of relation R to predicate t is the relation $\{(s, s')|t(s) \wedge (s, s') \in R\}$ (resp. $\{(s, s')|(s, s') \in R \wedge t(s')\}$). We admit without proof that the pre-restriction of a relation R to predicate t is $I(t) \circ R$ and the post-restriction of relation R to predicate t is $R \circ I(t)$. The *domain* of relation R is defined as $\delta(R) = \{s|\exists s' : (s, s') \in R\}$. We say that R is *deterministic* (or that it is a *function*) if and only if $\widehat{R}R \subseteq I$, and we say that R is *total* if and only if $I \subseteq R\widehat{R}$, or equivalently, $RL = L$, and *surjective* if and only if $LR = L$. A relation R is said to be *reflexive* if and only if $I \subseteq R$, *transitive* if and only if $RR \subseteq R$ and *symmetric* if and only if $R = \widehat{R}$.

4.2 Refinement Calculus

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

Definition 4.1 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, of course, to refining a pre/postcondition specification by weakening its precondition and/or strengthening its postcondition [12,23]. We abbreviate this property by $R \sqsupseteq R'$ or $R' \sqsubseteq R$. We submit that, modulo traditional definitions of total correctness [12,23], the following propositions hold.

- A program P is correct with respect to a specification R if and only if $[P] \sqsupseteq R$, where $[P]$ is the function defined by P.
- $R \sqsupseteq R'$ 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'. We admit without proof that the refinement relation is a partial ordering. In [2] Mili et al. analyze the lattice properties of this ordering and find the following results (See Figure 4):

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

We have a simple condition under which the join and meet take on special expressions; we submit this without proof in the proposition below.

Proposition 4.2 *If $RL = R'L = (R \cap R')L$ then R and R' have a join, given by the following formula: $R \sqcup R' = R \cap R'$. Then the meet of R and R' is given by the following formula: $R \sqcap R' = R \cup R'$.*

The condition of this proposition means that R and R' have the same domain, and for each element of their common domain, they have at least one image in common.

4.3 Approximating a Loop Function

We consider a while loop of the form: `while t do B` on some space S and we let W be the function of this loop; we assume that this loop terminates for all initial states in S (we have discussed in [20] in what sense this does not cause a significant loss of generality). Our stepwise approach to the derivation of the loop function is that we obtain this function by accumulating a sufficient number of (in)equations of the form $W \sqsupseteq T$, where T is some relation on S; we refer to T as a *lower bound* of W. By virtue of lattice properties of the refinement structure, if W refines T and T' then it refines their join. In practice, if we find a set of lower bounds $T_1, T_2, T_3, ...T_k$ to W, then we can infer: $W \sqsupseteq T_1 \sqcup T_2 \sqcup T_3 \sqcup ... \sqcup T_k$. By virtue of the structure of the refinement lattice (see Figure 4), if the join of all the T_i is total and deterministic, then it is maximal in the refinement ordering, whence

$$W \sqsupseteq T_1 \sqcup T_2 \sqcup T_3 \sqcup ... \sqcup T_k \Leftrightarrow W = T_1 \sqcup T_2 \sqcup T_3 \sqcup ... \sqcup T_k.$$

In such cases, we have found the function of the loop. If, on the other hand, the join of all the lower bounds we have found is not a total function, then we do not have the function of the loop, but we have an approximation of it. The following theorems are intended to provide us with lower bounds (in the refinement ordering) of the loop function. Due to lack of space, we do not present proofs of these threorems (the interested reader is referred to [20]), but may illustrate them with trivial examples.

Theorem 4.3 *We consider the while statement* `while t do B`*, where* $t \neq \textbf{false}$ *. Then*

$$T = I(t) \circ L \circ I(t) \circ [B] \circ I(\neg t) \cup I(\neg t)$$

is a lower bound for W*.*

A scrutiny of the relational expression of the lower bound reveals that it merely says that the final state of the loop satisfies $\neg t$, and its predecessor by B (when it exists) satisfies t.

Theorem 4.4 *If* R *is a reflexive transitive relation that is a superset of* $[B]$ *such that* $R \circ I(\neg t)$ *is total then* $T = R \circ I(\neg t)$ *is a lower bound of* W*.*

We consider the following while statement where x is a natural variable and a is a positive integer constant: `while x>=a {x=x-a;}` and we let W be the function of this loop. Theorem 4.4 mandates that we find a reflexive transitive superset of the relation defined by the loop body. We submit that $R = \{(x, x')|x \textbf{ mod } a = x' \textbf{ mod } a\}$ is reflexive (trivial), transitive (trivial), and that it is a superset of the loop body (if $x' = x + a$ then $x \textbf{ mod } a = x' \textbf{ mod } a$). We further find that $R \circ I(\neg t)$ is total. Whence we infer that W, the function of the loop, refines the following lower bound: $T = R \circ I(\neg t)$. We briefly evaluate this lower bound:

$$\begin{aligned}
& R \circ I(\neg t) \\
= & \qquad \{\text{substitution}\} \\
& \{(x, x')|x \textbf{ mod } a = x' \textbf{ mod } a \wedge x' < a\} \\
= & \qquad \{\text{simplification}\} \\
& \{(x, x')|x' = x \textbf{ mod } a\}.
\end{aligned}$$

Because this relation is total and deterministic, it is maximal in the refinement lattice; hence from $W \sqsupseteq T$ we infer $W = T$.

5 Detailed Algorithm

5.1 The Internal Representation

Because theorem 4.4 requires that we find a superset of the loop body, we must represent the loop body in a way that makes supersets visible. In typical programming languages, the loop body is represented as a sequence of statements, a structure which does not lend itself to finding supersets: in order to find the superset of a sequence, we must look at each term of the sequence. To obviate this difficulty, we propose to represent the loop body as an intersection instead of a sequence: indeed, if B is written as $B = B_1 \cap B_2 \cap B_3 \cap ... \cap B_n$, then a superset of B_1 is a superset of B, a superset of $B_1 \cap B_2$ is a superset of B, a superset of $B_1 \cap B_2 \cap B_3$ is a superset of B. The notation we have chosen to this effect is what is called

(Conditional) Concurrent Assignments, or *CCA*'s for short. These represent variable assignments that are carried out concurrently, or in an arbitrary order. Such a representation is obtained from a traditional sequential notation by removing all the sequential dependencies. For example, the sequence {`x=x+1; y=2*x;`} (notice the semi-colon separators) is transformed into {`x=x+1, y=2*x+2,`} (notice the comma separators). Generally, the assignments may be conditional (whence their name: Conditional Concurrent Assignments) because the loop body may contain if-then-else statements; but for the time being we do not consider conditionals, and briefly discuss if-then-else statements in section 6.

5.2 *Deriving Lower Bounds*

Once the loop body is structured in CCA form, we can derive lower bounds by looking at one statement at a time, or two statements at a time, or three statements at a time, etc. For the sake of controlling combinatorics, we resolve not to look at more than three statements at a time. To derive lower bounds of loop functions, we scan their loop body written in CCA form, match their statements or combinations of statements against pre-cataloged code patterns, and derive duly instantiated lower bounds in case of a match. We use the term *recognizer* to refer to the aggregate made up of variable declarations, code patterns, and corresponding lower bound; and we distinguish between *one-recognizers* that match one statement at a time, *two-recognizers* that match two statements at a time, *three-recognizers* that match three statements at a time. The current status of development of the extraction algorithm can be characterized by the following statements:

- All the machinery for recognizing code patterns and generating instantiated lower bounds is currently in place.
- We have a total of 28 recognizers, including ten 1-recognizers, fifteen 2-recognizers, and three 3-recognizers.

We can augment the scope of applicability of the algorithm by adding more recognizers, to handle new control structures and new data structures. Table 1 shows some sample recognizers that are currently implemented. The question of how recognizers are derived is beyond the scope of this paper; suffice it to say that they are derived using the concept of *strongest invariant functions* introduced in [21], and that they are discussed in greater detail in [20].

5.3 *Combining Lower Bounds*

Each recognizer produces (when it is successfully matched) a logic formula, which represents the relevant lower bound. In principle, we must now compute the join of all the lower bounds. However, all the lower bounds are total relations; by virtue of proposition 4.2, their join equals their intersection. In logical terms, this means that we take the conjunction ($\wedge$) of all the clauses that are generated by the recognizers. If this defines a total deterministic relation (a total function) then it is the function of the loop; else it is a lower bound of the function of the loop (i.e.

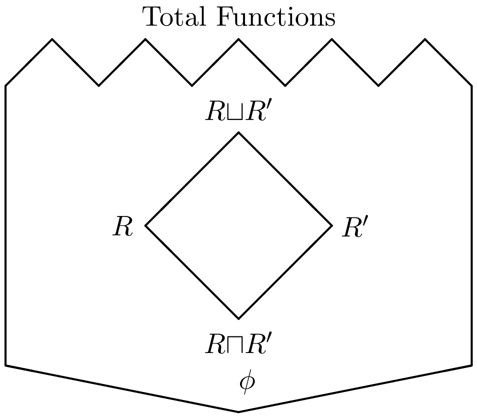

Fig. 4. Lattice Structure of Refinement

it specifies some, but not all, of the functional properties of the loop). In practice, if Mathematica returns an expression for each primed state variable (determinacy), and no restriction on the unprimed state variables (totality), then we have found the function of the loop

5.4 Illustration

For the sake of illustration, we consider the loop presented in section 2 and we present in turn excerpts of the loop written in the CCA format, then excerpts of the Mathematica file produced by the recognizers.

loop.cca:

```
{
const int a; const int b; const int c;
const int d; const int e; const int N;
const function f;
array int A; array int B;
list l;  list m;
int q; int qc;
int x; int y; int z; int t; int i;
int j; int v; int w; int sA; int sB;
while !(i == 0)
  {v = v+a*t, z = z+c*x, w = w+e*y,
   x = x+a, y = y+b, t = t*d,
   sA = sA+A[i], sB = sB+B[j],
   i = i-1, j = j+1, l = tail(l),
   m = m.head(l), q = f(q),
   qc = qc+q, A = A, B = B}
}
```

The algorithm produces 56 equations, of which we present the following excerpts:

ID	State Space	Code Pattern	Lower Bound $T =$
1R1	x: int const c: int >0	x=x+c	$\{(s,s')\|x \textbf{ mod } c =\}$ $x' \textbf{ mod } c\}$
1R2	x: int	x=x+1	$\{(s,s')\|x \leq x'\}$
1R3	x: int	x=x-1	$\{(s,s')\|x \geq x'\}$

2R1:	x, y: int
	const a, b: int
		x = x+a
		y = y+b
			$\{(s,s')\|ay - bx =$
			$ay' - bx'\}$

2R2:	x, y: int
	const a: int
		x = x*a
		y = y+x
			$\{(s,s')\|y(1-a) + x =$
			$y'(1-a) + x'\}$

2R3:	x, y: int
	const a, b: int
		x = x+a
		y = y*b
			$\{(s,s')\|\frac{y}{b^{x/a}} =$
			$\frac{y'}{b^{x'/a}}\}$

2R4:	x: listType
	y: listType
		y:=y.First(x)
		x:=Rest(x)
			$\{(s,s')\|$
			$y.x = y'.x'\}$

2R5:	i: int
	x: sometype
		i:=i-1,
		x:=f(x)
			$\{(s,s')\|$
			$f^i(x) = f^{i'}(x')\}$

3R1:	i: int
	x: sometype
	y: sometype
		i:=i-1,
		x:=f(x)
		y:=y+x
			$\{(s,s')\|$
			$y + \Sigma_{k=1}^{i} f^k(x) =$
			$y' + \Sigma_{k=1}^{i'} f^k(x')\}$

3R2	x: int
	a[N]: int
	i: int
		i=i+1,
		x = x+a[i]
		a=a
			$\{(s,s')\|a' = a$
			$\wedge x + \sum_{k=i}^{N} a[k] =$
			$x' + \sum_{k=i'}^{N} a'[k]\}$

3R2	x: int
	a[N]: int
	i: int
		i=i-1,
		x = x+a[i]
		a=a
			$\{(s,s')\|a' = a$
			$\wedge x + \sum_{k=1}^{i} a[k] =$
			$x' + \sum_{k=1}^{i'} a'[k]\}$

Table 1
1-, 2-, and 3-Recognizers

loop.mat

```
Reduce[ Reduce[ {
Mod[x,Abs[a]]==Mod[xP,Abs[a]],
Mod[y,Abs[b]]==Mod[yP,Abs[b]],
4.  Mod[t,Abs[Log[d,10]]]==Mod[tP,Abs[Log[d,10]]],
5.  Mod[i,Abs[1]]==Mod[iP,Abs[1]],
6.  i>=iP,
7.  Mod[j,Abs[1]]==Mod[jP,Abs[1]],
8.  j<=jP,
9.  A==AP,
10. B==BP,
11. v+a*t/(1-d)==vP+a*tP/(1-d),
12. z-c*x*(x-a)/(2*a)==zP-c*xP*(xP-a)/(2*a),
13. w-e*y*(y-b)/(2*b)==wP-e*yP*(yP-b)/(2*b),
14. a*y-b*x==a*yP-b*xP,
15. b*x-a*y==b*xP-a*yP,
16. t/d^(x/a)==tP/d^(xP/a),
17. a*i+1*x==a*iP+1*xP,
18. a*j-1*x==a*jP-1*xP,
19. 1*x-a*j==1*xP-a*jP,
20. t/d^(y/b)==tP/d^(yP/b),
21. b*i+1*y==b*iP+1*yP,
22. b*j-1*y==b*jP-1*yP,
23. 1*y-b*j==1*yP-b*jP,
24. t/d^(j/1)==tP/d^(jP/1),
25. 1*i+1*j==1*iP+1*jP,
26. lP==Nest[Rest,l,i-iP],
27. i-Length[l]==iP-Length[lP],
28. Nest[f,q,i]==Nest[f,qP,iP],
29. lP==Nest[Rest,l,jP-j],
30. j+Length[l]==jP+Length[lP],
31. Join[m,l]==Join[mP,lP],
32. sA+Sum[A[k], {k,1,i}]==sAP+Sum[AP[k], {k,1,iP}],
33. sB+Sum[B[k], {k,j,N}]==sBP+Sum[BP[k], {k,jP,N}],
34. qc+Sum[Nest[f,q,k],{k,1,i}]==qcP+Sum[Nest[f,qP,k],{k,1,iP}],
35. (iP==0),
36. Exists [ {APP,BPP,iPP,jPP,lPP,
        mPP,qPP,qcPP,sAPP,sBPP,tPP,vPP,wPP,xPP,yPP,zPP},
37. !(iPP==0) && ... ... ...
38. zP==zPP+c*xPP &&
39. vP==vPP+a*tPP]
40. }],
41. {iP, jP, lP, mP, qP, qcP, sAP,
42. sBP, tP, vP, wP, xP, yP, zP},
43. Backsubstitution->True]
```

Lines 1 and 43 are Mathematica instructions/ options. Lines 41 and 42 specify that we want the given equations resolved in these variables, which are the final values of the program variables. Lines 2 to 8 represent the application of 1-recognizers. Lines

9 to 31 represent the application of 2-recognizers. And lines 32 to 34 represent the application of the 3-recognizers. Line 35 represents the clause $\neg t(s')$ that we have factored out from all the lower bounds. Lines 36 to 40 represent the application of theorem 4.3 (of which we have deleted many clauses). This theorem specifies that the state immediately preceding the final state (represented by PP) satisfies the loop condition t; in other words, the final state (specified by P) is the first state that fails to satisfy the loop condition.

6 Assessment and Prospects

6.1 Handling Conditionals

Our divide-and-conquer approach is heavily dependent on writing the loop body as an intersection of concurrent assignments. The introduction of conditionals (if-then, if-then-else) compromises this regular structure by introducing union operators between the assignments. In order to find a superset of a union, one has to look at both terms of the union, which is at odds with our divide-and-conquer philiosophy, that advocates localized inspections. The theorem below allows us to derive a lower bound of the loop function in the presence of if-then-else statements, without having to look at their then-branch and else-branch simultaneously, but rather in turn.

Theorem 6.1 *We consider a while statement of the form*

```
while t do if u then P else Q
```

on space S that terminates for all s in S and we let W be the function of this while statement. If R and R' are reflexive transitive relations such that

$$I(u) \circ [P] \subseteq R,$$

$$I(\neg u) \circ [Q] \circ R \subseteq R'$$

and

$$R \circ R' \circ I(\neg t) \circ L = L$$

then

$$W \sqsupseteq R \circ R' \circ I(\neg t)$$

i.e. $T = R \circ R' \circ I(\neg t)$ *is a lower bound for W.*

As an illustration of this theorem, we consider the following loop on natural variables x, y, z:

```
w =
while !(y==0)
   {if (y%2 == 1)
      {y = y-1; z = z+x;}
   else
      {x = 2*x; y = y/2;}
   }
```

We let P and Q be defined as the functions of (respectively) the then branch and the else branch of the `if-then-else` statement in the loop body. We find,

$$P = \{(s, s')|y \textbf{ mod } 2 = 1 \wedge x' = x \wedge y' = y - 1 \wedge z' = z + x\},$$

$$Q = \{(s, s')|y \textbf{ mod } 2 = 0 \wedge x' = 2 \times x \wedge y' = y/2 \wedge z' = z\}.$$

We propose the following reflexive transitive relation that is a superset of P:

$$R = \{(s, s')|z + x \times y = z' + x' \times y'\}.$$

For R', we take the following superset of QR:

$$R' = \{(s, s')|z + x \times y = z' + x' \times y'\}.$$

According to theorem 6.1, $R \circ R' \circ I(\neg t)$ is a lower bound for $[w]$. We find the following relation

$$\{(s, s')|z' = z + x \times y \wedge y' = 0\},$$

which is a lower bound for $[w]$, as the reader can verify.

6.2 Related Work

Our work is related to three lines of research: 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 [9]. 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.

Generally, the derivation of loop invariants is closely related to the derivation of loop functions since they both aim to discover the inductive argument that underlies the behavior of the loop. Furthermore, a theorem by Mills [22] shows how loop functions can be used to produce loop invariants. Also, the generation of lower bounds that we carry out to approximate the function of a loop is reminiscent of the extensive work that has been done and is being done on generating loop invariants [15]. Many researchers in the theorem proving and the program verification communities have lent much attention to the goal of extracting loop invariants [3,25,5,24,17,4,7,16,6,18,26]. In [10] 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. In [8], Denney and Fischer analyze generated code against safety properties, for the purpose of certifying the code. To this effect, they proceed by matching the generated code against known idioms of the code generator, which they parametrize with relevant safety properties. Safety properties are formulated by invariants (including loop invariants), which are inferred by propagation through the code. In [5], Colón et al. consider loop invariants of numeric programs as linear expressions and derive the coefficients of the expressions by solving a set of linear equations; they extend this work to non linear expressions in [24]. In [17] Kovacs and Jebelean derive loop invariants by solving recurrence relations; they pose the loop invariants as solutions to recurrence relations, and derive closed forms of the solution using a theorem prover (Theorema) to support the process. In [3] Rodriguez Carbonnell et al. derive loop invariants by forward propagation and fixed point computation, with robust theorem proving support; they represent loop bodies as conditional concurrent assignments, whence their insights are of interest to us as we envision to integrate conditionals into our concurrent assignments. In [19], we discuss the difference between traditional loop invariants (in the sense of Hoare's logic [13,12]) and the loop invariants that we derive in this paper from invariant functions, which we call *reflexive transitive loop invariants*. Less recent work on loop invariants includes work by Cheatham and Townley [4], Karr [16], Cousot and Halwachs [7], and Mili et al [21]. Work on loop analysis and loop transformations in the context of compiler construction is also related to functional extraction, although to a lesser degree than work on loop invariants [11,1].

In [14] 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).

7 Conclusion

The goal of computing program functions, notably for iterative programs, is a difficult goal, but is nevertheless a worthwhile goal, given the advances that it affords us in terms of program comprehension, program analysis, reverse engineering, software maintenance, software inspection, etc. In this paper, we have presented some mathematical results, and have shown their preliminary application to the derivation of an algorithm for computing loop functions; also, we have illustrated the behaviour of the algorithm on a simple example. The current algorithm has all the necessary infrastructure to derive Mathematica equations; the capability of the algorithm evolves through the addition of new recognizers. In the short term, the bottleneck of this process is that we can only generate symbolic equations that Mathematica can resolve. Yet new application domains involve domain-specific knowledge, whose integration requires an inference capability; we are not sure yet whether Mathematica can fulfill this need. Another bottleneck, that may arise in the medium term as the number of recognizers grows, is the need to control redundancy; while we have many ideas on how to do this, they are all likely to significantly increase the complexity of the algorithm. An equally pressing need, of course, is the ability to deal with conditionals; we have a theorem (not presented in this paper, but alluded to) that supports this step, using relational identities. We fully expect such a solution to increase the complexity of the algorithm; in particular, it will involve a more intensive interaction between the recognizer-based matching and the symbolic equation manipulation of Mathematica.

On balance, we argue that the proposed approach is worthy of further investigation, as it takes an angle to the analysis of while loops that is fairly orthogonal to existing approaches, and is likely to complement their results and their insights.

References

[1] U. Banerjee. *Loop Transformations for Restructuring Compilers.* Kluwer Academic Publishers, Boston, MA, 1993.

[2] N. Boudriga, F. Elloumi, and A. Mili. The lattice of specifications: Applications to a specification methodology. *Formal Aspects of Computing*, 4:544–571, 1992.

[3] E. R. Carbonnell and D. Kapur. Program verification using automatic generation of invariants. In *Proceedings, International Conference on Theoretical Aspects of Computing '2004*, volume 3407, pages 325–340. Lecture Notes in Computer Science, Springer Verlag, 2004.

[4] T. E. Cheatham and J. A. Townley. Symbolic evaluation of programs: A look at loop analysis. In *Proc. of ACM Symposium on Symbolic and Algebraic Computation*, pages 90–96, 1976.

[5] M. A. Colon, S. Sankaranarayana, and H. B. Sipna. Linear invariant generation using non linear constraint solving. In *Proceedings, Computer Aided Verification, CAV 2003*, volume 2725 of *Lecture Notes in Computer Science*, pages 420–432. Springer Verlag, 2003.

[6] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings, Fourth ACM Symposium on Principles of Programming Languages*, Los Angeles, CA, 1977.

[7] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the Fifth Annual ACM SIGPLAN-SIGACT Symposium on the Principles of Programming Languages*, pages 84–97, 1978.

[8] E. Denney and B. Fischer. A generic annotation inference algorithm for the safety certification of automatically generated code. In *Proceedings, the Fifth International Conference on Generative programming and Component Engineering*, Portland, Oregon, 2006.

[9] D. Dunlop and V. R. Basili. A heuristic for deriving loop functions. *IEEE Transactions on Software Engineering*, 10(3):275–285, May 1984.

[10] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 2006.

[11] T. Fahringer and B. Scholz. *Advanced Symbolic Analysis for Compilers.* Springer Verlag, Berlin, Germany, 2003.

[12] D. Gries. *The Science of programming.* Springer Verlag, 1981.

[13] C. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576 – 583, Oct. 1969.

[14] L. Hu, M. Harman, R. Hierons, and D. Binkley. Loop squashing transformations for amorphous slicing. In *Proceedings, 11th Working Conference on Reverse Engineering*. IEEE Computer Society, 2004.

[15] T. Jebelean and M. Giese. *Proceedings, First International Workshop on Invariant Generation*. Research Institute on Symbolic Computation, Hagenberg, Austria, 2007.

[16] M. Karr. Affine relationships among variables of a program. *Acta Informatica*, 6:133–151, 1976.

[17] T. J. L. Kovacs. An algorithm for automated generation of invariants for loops with conditionals. In D. Petcu, editor, *Proceedings of the Computer-Aided Verification on Information Systems Workshop (CAVIS05), 7th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC05)*, pages 16–19, Department of Computer Science, West University of Timisoara, Romania, 2005.

[18] T. Marlowe and B. Ryder. Properties of dataflow frameworks: A unified model. *Acta Informatica*, 28:121–163, 1990.

[19] A. Mili. Reflexive transitive loop invariants: A basis for computing loop functions. In *First International Workshop on Invariant Generation*, Hagenberg, Austria, June 2007.

[20] A. Mili, S. Aharon, M. Pleszkoch, and R. Linger. Towards the automated derivation of loop function. Technical report, NJIT, http://web.njit.edu/m̃ili/fxloop.pdf, September 2007.

[21] A. Mili, J. Desharnais, and J. R. Gagne. Strongest invariant functions: Their use in the systematic analysis of while statements. *Acta Informatica*, April 1985.

[22] H. Mills. The new math of computer programming. *Communications of the ACM*, 18(1), January 1975.

[23] C. Morgan. *Programming from Specifications*. International Series in Computer Sciences. Prentice Hall, London, UK, 1998.

[24] S. Sankaranarayana, H. B. Sipna, and Z. Manna. Non linear loop invariant generation using groebner bases. In *Proceedings, ACM SIGPLAN Principles of Programming Languages, POPL 2004*, pages 381–329, 2004.

[25] B. Scholz and T. Fahringer. *Advanced Symbolic Analysis of Compilers*. Springer Verlag, 2003.

[26] M. Sharir and A. Pnueli. Two approaches to inter procedural data flow analysis. In Jones and Muchnik, editors, *Program Flow Analysis: Theory and Applications*, 1981.