Deriving Loop Functions from Reflexive Transitive Loop Invariants

Rahma Ben Ayed[1], Ali Mili[2], Chaitanya Nadkarni[2], and Mustafa Korkmaz[2]

[1] ENIT, University of Tunis El Manar
Tunis, 1002 Tunisia, rahma.benayed@enit.rnu.tn
[2] NJIT, College of Computer Science, Newark NJ 07102
cgn4@njit.edu, mxk4824@njit.edu, mili@cs.njit.edu

Abstract. Most activities of program comprehension can be understood, ultimately, as efforts to derive the function that a program or program part defines on its state space, or from its inputs to its outputs. Among traditional programming language constructs, iteration is one of the most difficult to analyze, because the derivation of the function of a loop requires the invention of an inductive argument, a creative step that does not lend itself to easy automation. In this paper, we briefly present some theoretical results from a refinement calculus, which we deploy to derive the function of while loops in a stepwise manner, by successive approximations. We discuss our current effort in deploying the proposed approach to develop an automated tool that computes the function of while loops under some restrictive conditions. We also discuss how to expand the capability of the proposed tool.

Keywords

Function extraction, relational calculus, refi nement calculus, computing loop behavior, relational algebras, loop functions, loop invariants, reflexive transitive loop invariants.

1 Introduction

1.1 Deriving Loop Functions

Denotational semantics interpret a program or program part by the function that this program (or program part) computes on its space [19]; the space of the program is, in turn, defi ned by the aggregate of variable declarations that the program manipulates. The traditional control structures of programming languages can be interpreted by means of relational operators: the sequence statement (;) is interpreted by functional composition; the alternation statement (if-then-else) is interpreted by functional union; the conditional (if-then) is interpreted by functional restriction (and union); and iteration (while-do) is interpreted by transitive closure.

It is possible to imagine an automated tool computing the function of a complex program in a stepwise bottom-up fashion, so long as the only control structures it encounters are sequence, conditional, and alternation. In order for such a tool to handle while loops, it needs to be able to bridge the *inductive gap* between the function of the loop body and the function of the whole loop. It is not suffi cient to say that the function of the while loop is the transitive closure of the function of the loop body: in most cases that does not give useful, explicit information to the analyst. The purpose of this paper is to explore a refi nement-based stepwise technique for deriving loop functions. Our approach is based on the following premises:

- *A Closed Form of the Loop Function*. We need to fi nd a closed form representation of the loop function, one that gives the fi nal states of the program variables as a function of the initial states.
- *A Divide and Conquer Approach*. We envision that the function of the loop is derived without ever having to deal with the functional details of the loop all at once. Instead, we derive the function of the loop in a stepwise fashion, by successive approximations, each approximation depending on analyzing arbitrarily little information about the loop body.
- *Ability to Approximate the Loop Function*. The successive approximations alluded to in the previous premise produce a monotonic sequence of increasingly accurate characterizations of the loop function, obtained by collecting independent functional details about the loop. We envision that even if we are unable to codify all the relevant functional information of the loop, we be able to provide some useful characterization of the loop function.

– *A Refinement Based Approach*. We use the lattice structure of the refinement ordering as the basis for our approach. In this approach, we derive the function of the loop from a set of independent inequalities involving the loop function; we present in section 3.3 a simple example that illustrates, by analogy, the main idea of our approach.

One may ask: why is it important to derive the function of a loop, or the function of any program, and why is this important now? We put forth a number of premises to argue this case.

– In an era when software is used in life-critical, mission-critical, and safety-critical applications, it is no longer sufficient to inspect source code or to test it, in order to meet quality standards. Rather we need dependable, certifiable means to analyze program functions in all their functional detail, in all circumstances of use, under all operational conditions.
– In an era of pervasive malicious code and heightened security concerns, it is no longer sufficient to ask the question: Does the program do what we want? We must also ask the question: What else does the program do, that perhaps we do not want? The first question can be answered by verifying that the program refines some specification (representing the desirable effect); the second question can be answered by verifying that the program function does not refine some specification (representing the undesirable effect). In both cases, we need means to capture the function of the program in its most minute detail.
– In an era of code outsourcing, COTS-based development, and code reuse, where we are increasingly relying on code we have not written and had no control over its development process, it is no longer sufficient to depend on (punctual) certification testing to ensure product quality. Rather we need means to formulate and certify general claims about the functional properties of programs.
– In an era when software professionals continue to spend much of their time analyzing and understanding code, the availability of tools that help them derive program functions is likely to have a significant impact on the economics of software engineering [5, 14].

1.2 Reflexive Transitive Loop Invariants

The analysis of loop functions that we present in this paper is, justifiably, reminiscent of research on loop invariants [1, 3, 4, 6–8, 11, 12, 16–18, 22, 26]. We briefly explore the difference between traditional loop invariants, used as inductive assertions [15], and the loop invariants we use in this paper to derive loop functions. First, we consider a simple illustrative example involving a loop that computes the sum of an array.

```
x:  xtype; i:  1..N+1;
a:  array [1..N] of xtype;

{x= 0;  i= 1;
 while (i != N+1)
   {x= x+a[i];  i= i+1}}
```

For this loop, we let the precondition and postcondition be defined as:

$$\phi(s_0, s) \equiv a = a_0 \wedge x = 0 \wedge i = 1,$$

$$\psi(s_0, s) \equiv x = \sum_{k=1}^{N} a_0[k].$$

An adequate invariant for this specification is:

$$\chi(s_0, s) \equiv a = a_0 \wedge x = \sum_{k=1}^{i-1} a[k].$$

According to [15], in order to prove that the while statement is partially correct with respect to the specification $(\phi(s_0, s), \psi(s_0, s))$, it suffices to prove the following premises.

1. Initial condition:
$$\phi(s_0, s) \Rightarrow \chi(s_0, s).$$
2. Invariance (Inductive) condition:
$$\{\chi(s_0, s) \wedge t\}\ B\ \{\chi(s_0, s)\}.$$
3. Exit (Final) condition:
$$\chi(s_0, s) \wedge \neg t \Rightarrow \psi(s_0, s).$$

We leave it to the reader to check that these three premises hold for the specification and the loop invariant at hand.

Now, we consider the following reflexive transitive relation

$$R = \{(s, s') | a = a' \wedge x + \sum_{k=i}^{N} a[k] = x' + \sum_{k=i'}^{N} a'[k]\},$$

and we argue that this relation defines a loop invariant in the following sense: the predicate χ defined by

$$\chi(s_0, s) \equiv (s_0, s) \in R$$

satisfies the second premise of Hoare's rule, i.e.

$$\{\chi(s_0, s) \wedge t\}\ B\ \{\chi(s_0, s)\}.$$

We leave it to the reader to check that the proposed predicate satisfies Hoare's invariance condition. Note however that the relation R that we propose is more general than the invariant predicate χ, in the sense that the latter is obtained by fixing a parameter (s instantiated to s_0) of the former. In [21] we have explored the differences between these forms of invariants, and have shown in particular different formulas for the strongest invariant assertion, and the strongest reflexive transitive loop invariant. A detailed discussion of these formulas is beyond the scope of this paper; we content ourselves in this paper with a brief informal discussion of the main differences between traditional loop invariants (used in inductive arguments on the correctness of a loop with respect to a pre-condition/ post-condition pair) and the reflexive transitive loop invariants that we introduce in this paper for the purpose of computing loop functions.

– *Different Arities*. Traditional inductive assertions have the form $\phi(s_0, s)$, where s_0 represents the initial state; whereas the proposed loop invariants have the form $(s, s') \in R$, where s and s' are both arbitrary states. Hence the former are unary predicates, whereas the latter are binary predicates. In the sequel, we refer to them respectively as *unary invariants* and *binary invariants*; also, the terms *binary loop invariant* and *reflexive transitive loop invariant* will be used to the same concept, though stressing different attributes.
– *Different Dependencies*. A far more important distinction is that a unary loop invariant depends not only on the loop under consideration but also on the specification (precondition/ postcondition pair). By contrast, a binary loop invariant depends exclusively on the loop under consideration, regardless of its context. This is why, in the example presented above, we checked the second condition of Hoare's method, i.e.

  $$\{(s_0, s) \in R \wedge t\}\ B\ \{(s_0, s) \in R\}$$

  but we did not check the initial condition

  $$\phi(s_0, s) \Rightarrow (s_0, s) \in R,$$

  nor the final condition

  $$(s_0, s) \in R \wedge \neg t \Rightarrow \psi(s_0, s).$$

  The binary loop invariant does not depend on the precondition and the postcondition, hence is not subject to any equation that involves them. An immediate consequence of this difference, is that if we change the specification or the initialization of the loop we have to change its unary invariant, but we do not have to change its binary invariant.

– *Different Levels of Abstraction*. Specifically, we argue that binary loop invariants subsume unary loop invariants, in the following sense: from any binary loop invariant (R), we can generate a unary loop invariant (χ) that satisfies Hoare's invariance condition
$$\{\chi() \wedge t\}\ B\ \{\chi()\}.$$
– *Different Methods*. Due to their unary nature, unary loop invariants can be derived and analyzed by induction on the trace of execution of the loop: if the invariant holds up to a point in the execution trace, then it holds after one more execution of the loop body. Because binary loop invariants juggle two arguments (s and s'), a simple induction on the execution trace does not do them justice; then we recourse to an induction on the loop structure (if the function of the loop body satisfies this property, then the function of the loop satisfies that property).

Intuitively, we can interpret the contrast between unary loop invariants and binary loop invariants in the following terms: whereas a unary loop invariant expresses a condition that holds for the current state after an arbitrary number of iterations starting from a fixed initial state, a binary loop invariant expresses a condition that holds between a state s and a state s' given that s' appears an arbitrary number of iterations after s. We often encounter loop invariants that are not only reflexive and transitive, but also symmetric: for those, the roles of s and s' are interchangeable, anyone of them may precede the other, all we know is that they are an arbitrary number of iterations apart. A more detailed discussion of the contrast between unary invariants and binary invariants is given in [21].

In the next section, we introduce some mathematical background, which we use in section 3 to present some theorems that elucidate how loop functions can be derived from binary loop invariants. Then in section 4 we discuss how binary loop invariants can themselves be derived from a stepwise analysis of the source code of a loop. This is followed in section 5 by a brief discussion of a tool we are currently developing to automatically generate the function of loops written in (a limited subset of) C/ C++/ Java. Finally, in the conclusion we summarize our finding and sketch plans for extending our work.

2 Mathematical Background

2.1 Elements of Relations

We represent the functional specification of programs by relations; without much 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 $x(s')$ (and do the same for other variables).

As a specification, a relation contains all the (input,output) pairs that are considered correct by the specifier. 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 $\widehat{R}$ or $R\widehat{\ }$, 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 $dom(R) = \{s|\exists s' : (s,s') \in R\}$. The *range* of relation R is denoted by $rng(R)$ and defined as $dom(\widehat{R})$. The *nucleus* of relation R is the relation denoted by $\mu(R)$ and defined by $R\widehat{R}$. For any R, the nucleus of R is symmetric and reflexive on $dom(R)$. 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, $RL = L$.

Given a relation R on S and an element s in S, we let the *image set* of s by R be denoted by $s.R$ and defined by $s.R = \{s'|(s,s') \in R\}$. 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 $I \subseteq R$, *transitive* if and only if $RR \subseteq R$ and *symmetric* if and only if $R = \widehat{R}$. We will occasionally refer to *Tarski's Identity* [27, 28], which provides that for all relation R, $LRL = L$ if and only if R is non-empty.

We are interested in two special types of rectangular relations: rectangular surjective relations are called *vectors* and satisfy the condition $RL = R$; rectangular total relations are called *invectors* (inverse of a vector) and satisfy the condition $LR = R$. In set theoretic terms, a vector on set S has the form $A \times S$, and an invector has the form $S \times A$, for some subset A of S. Vector $A \times S$ can also be written as $I(A) \circ L$.

2.2 Relations Based Refinement

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, of course, to refining a pre/postcondition specification by weakening its precondition and/or strengthening its postcondition [13, 25]. We abbreviate this property by $R \sqsupseteq R'$ or $R' \sqsubseteq R$. We admit that, modulo traditional definitions of total correctness [9, 13, 20], 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 any relation R can be refined by a deterministic relation, i.e. a function.

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:

- 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.

See Figure 1.

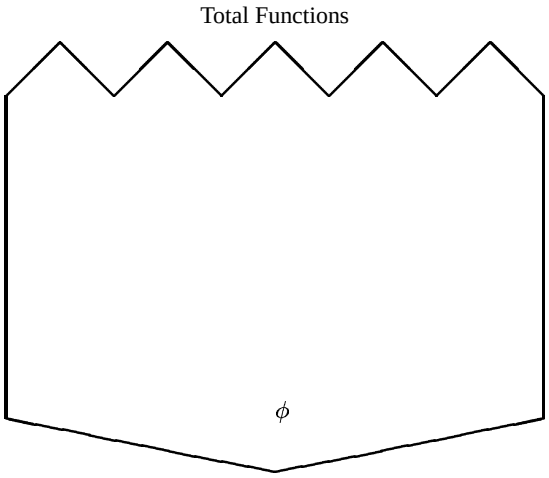

Fig. 1. Lattice Structure of Refinement

3 Deriving Loop Functions from Reflexive Transitive Loop Invariants

3.1 Theorem of Binary Loop Invariants

The interest of binary loop invariants is reflected in the following theorem, due to [23].

Theorem 1. *We consider a while loop on space S of the form* `w = while t do B`. *If R is a binary loop invariant for w, then the loop function W refines the following expression:*

$$R \cap \widehat{\overline{T}}$$

where T is the vector defined by predicate t.

In other words, if R is a binary loop invariant for w, then we can infer

$$W \sqsupseteq R \cap \widehat{\overline{T}}.$$

Interpretation of this theorem: The function of the loop is actually given by the following expression:

$$W = (T \cap [B])^* \cap \widehat{\overline{T}},$$

where $(T \cap [B])^*$ is the reflexive transitive closure of $(T \cap [B])$, i.e. the smallest reflexive transitive relation that is a superset of $(T \cap [B])$. Of course, in practice, it is very diffi cult in general to derive the transitive closure of an arbitrary function. What this theorem does is to strike a deal with us:

- Rather than ask us to derive the smallest superset of $(T \cap [B])$ that is reflexive and transitive, it asks us for a binary loop invariant of w, which is an arbitrarily large superset of $(T \cap [B])$ that is reflexive and transitive.
- On the other hand, rather than provide us with the exact function of the loop, it provides us with a lower bound (in the refi nement ordering) of the loop function.

This theorem enables us to derive the loop function by deriving arbitrary (arbitrarily large/ weak) binary loop invariants, then combining them (by the lattice operation of *join*) to obtain the loop function (or an approximation thereof). For example, if we have established:

$$W \sqsupseteq V_1,$$

$$W \sqsupseteq V_2,$$

then we can infer

$$W \sqsupseteq V_1 \sqcup V_2.$$

3.2 Rectangular Lower Bounds

While theorem 1 provides a lower bound of the loop function by means of a reflexive transitive loop invariant, the following two theorems (due to [23]) provide lower bounds in terms of rectangular relations. Relying exclusively on reflexive transitive lower bounds is not sufficient, as it fails to take into account termination conditions, such as

- The fact that the final state does not satisfy condition t, and
- The fact that the penultimate state does satisfy condition t.

The following theorems (whose proof is given in [23]) capture these two conditions.

Theorem 2. *Let w be the while loop defined by* `while t do B`. *If $t \not\equiv$* **false** *then*

$$[w] \sqsupseteq V$$

where $V = I(t) \circ L \circ I(t) \circ [B] \circ I(\neg t) \cup I(\neg t)$.

This theorem provides a lower bound for $[w]$ that says, in effect, that the final state satisfies $\neg t$, but also that the antecedent of the final state by $[B]$ satisfies t (i.e. that the final state is the first state to satisfy $\neg t$ in the sequence of successive states produced by the execution of the loop). This theorem is used whenever the condition $\neg t$ is not sufficient to determine the final state. For example, if we consider the loop

```
while i<>0 do i:= i-1
```

then we know that at the end of the execution, $i = 0$. But if the loop were written as:

```
while i>0 do i:= i-1
```

then not only do we know that at the end of the execution, $i \leq 0$, we also know (by virtue of theorem 2) that $i + 1 > 0$; from which we infer $i = 0$.

Theorem 3. *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 $t \not\equiv$* **false** *. Then*

$$[w] \sqsupseteq V$$

where V is defined as:

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

The lower bound provided by this theorem may be useful in cases where the function of the loop body ($[B]$) is not surjective. If it is surjective, then $L \circ [B] = L$, whence $V = L \circ I(\neg t)$, which merely expresses that the final state satisfies $\neg t$ (which is redundant with the other theorems).

3.3 A Numeric Analogy

All the theorems we have presented in this paper provide lower bounds for the function of the loop. We submit that this may be sufficient to derive the function of the loop, on account of the following analogy. We consider a lattice structure that has the same properties as the lattice of refinement, and we show how we can compute an unknown by means of a sufficient number of lower bounds, and how we can approximate it in cases where the lower bounds do not allow us to compute it with precision. Specifically, we consider the set of positive natural numbers included between 1 and 2000, and we consider the *divisible-by* relation between such numbers, which is a partial ordering: Any two numbers in S have a greatest lower bound (the GCD); not all pairs have a least upper bound (the smallest common multiple of 300 and 301, for example, is not in S); the set has a universal lower bound (which is 1, that divides every element of S); the set

has no universal upper bound; all numbers between 1001 and 2000 are maximal. This is the same structure as the lattice of refinement, where: any two specifications have a greatest lower bounds; least upper bounds exist conditionally; the lattice has a universal lower bound (the empty specification); the lattice has no universal upper bound; total deterministic relations are maximal elements in the lattice.

Imagine having to derive a number X in S given that we know the following properties about it: X is divisible by 15, X is divisible by 21, X is divisible by 33, X is divisible by 35, X is divisible by 55. From all these claims, we infer that X is divisible by the least common multiple of 15, 21, 33, 35 and 55, which is 1155. Because 1155 is maximal, the only number that is divisible by 1155 is 1155 itself. Hence $X = 1155$.

If all we knew about X were that X is divisible by 15, 33, and 55, then all we could infer would be that X is divisible by 165, i.e. that X is in the set

$$\{165, 330, 495, 660, ...1980\}.$$

Likewise, the approach we present in this paper derives the function of the loop by combining partial refinement claims of the form: the loop function refines this lower bound. If the join of all the lower bounds does not produce a maximal element (total deterministic relation), then we cannot find the function of the loop, but we have a specification that is refined by the loop.

4 Deriving Reflexive Transitive Loop Invariants

Whereas theorems 2 and 3 are constructive, in the sense that they provide an explicit expression of a lower bound of the loop function, theorem 1 is not constructive. Rather, it asks us to derive a reflexive transitive loop invariant, and once we do that it uses it to build a lower bound for the loop function. In this section, we briefly consider a pattern-matching method that derives reflexive transitive loop invariants by inspection.

4.1 Conditional Concurrent Assignments

As we recall from section 1, one of the premises of our approach is that we want to derive the function of the loop in a stepwise manner, by inspecting arbitrarily small portions of the code at a time. If we consider a few lines of sequential C/ C++/ Java code, such as

```
x:= x+1;
y:= 2*x;
z:= z+y
```

where x, y and z are integer variables, for example, then looking at one line at a time does not tell us much about the function of the loop body. If we consider line `y:= 2*x` for example, we cannot infer that the loop body refines the following relation,

$$\{(s, s')|y' = 2 \times x\}$$

since we do not know how the loop body may have changed x previously, nor how it may change y subsequently. Hence the only way to analyze the loop in a stepwise manner is to transform the sequential code into concurrent assignments, where each assignment summarizes the effect of the loop body on a particular variable. In the example above, the loop body can be rewritten as the following set of concurrent assignments:
$x := x + 1$,
$y := 2 \times (x + 1)$,
$z := z + 2 \times (x + 1)$.
Now each line tells us a complete story, and can be analyzed in isolation to derive a lower bound for the loop function. Hence for the sake of separation of concerns, we apply the analysis techniques on concurrent assignments rather than sequential assignments. If the loop body had conditional statements (if-then) or alternative statements (if-then-else) these concurrent assignments could be conditional concurrent assignments. In the scope of this paper, we limit ourselves to simple concurrent assignments, admitting a massive loss of generality, and deferring the treatment of conditionals to future work.

4.2 Deriving Lower Bounds

Theorem 1 tells us how to derive a lower bound of the loop function once we have a reflexive transitive loop invariant, but we still need means to derive reflexive transitive loop invariants to use it. In the sequel, we briefly present a pattern recognition method that derives reflexive transitive loop invariants by matching concurrent assignments of the loop body against specifi c patterns, and infers a lower bound whenever a match is successful.

To this effect, we use a repository of *recognizers*, where a recognizer is characterized by its state space, the pattern of concurrent statements it recognizes, and the lower bound that it provides for $[w]$. Hence the derivation of the loop function may proceed by matching parts of the loop body, written as a set of concurrent assignments, against existing statement patterns, and producing lower bounds for $[w]$ in case of a match. This algorithm has at its disposal a database of recognizers, which it scans starting with 1-Recognizers (that match one assignment statement), then 2-Recognizers (that match combinations of two statements), then 3-Recognizers (that match triplets). To keep the combinatorics tractable, we limit ourselves to recognizers whose length does not exceed 3 for the time being.

4.3 Sample 1-Recognizer

Generally, 1-Recognizers answer the question: what can we infer about the loop function if we know that this statement (in the loop body) gets executed an arbitrary number of times? We present and illustrate a sample 1-Recognizer given in Figure 2. For illustration, let's consider a while loop whose loop body is written as a set of concurrent assignments, as follows:

```
while y>0 do
   {... ... ...
   x:= x+c,
   ... ... ...}
```

where x is an integer variable and c is an integer constant greater than 0. Application of this sample recognizer provides that $[w]$ refi nes the following specifi cation:

$$V = \{(s, s') | x \bmod c = x' \bmod c \wedge y' \leq 0\}.$$

Note that we could make this claim on the loop function using very little information on the loop, regardless of what the ellipsis in the loop body stands for.

State Space	Semantic Pattern	Lower Bound
x: int const c: int >0	x:=x+c	$V =$ $\{(s, s') \| x \bmod c = x' \bmod c \wedge \neg t(s')\}$

Fig. 2. Sample 1-Recognizer

State Space	Semantic Pattern	Lower Bound
x: listType y: listType	y:=y.head(x) x:=tail(x)	$V =$ $\{(s, s') \| x.y = x'.y' \wedge \neg t(s')\}$
i: int x: sometype	i:=i-1, x:=f(x)	$V =$ $\{(s, s') \| f^i(x) = f^{i'}(x') \wedge \neg t(s')\}$

Fig. 3. Sample 2-Recognizer

State Space	Semantic Pattern	Lower Bound
i: int x: sometype a: sometype	i:=i-1, x:=f(x) a:=a+x	$V =$ $\{(s,s')\|f^i(x) = f^{i'}(x') \wedge$ $a + \Sigma_{k=1}^{i} f^k(x) = a' + \Sigma_{k=1}^{i'} f^k(x') \wedge \neg t(s')\}$

Fig. 4. Sample 3-Recognizer

4.4 Sample 2-Recognizers

Generally, 2-Recognizers answer the question: what can we infer about the loop function if we know that these two statements get executed the same number of times? We present and illustrate two sample 2-Recognizers given in Figure 3, where `head` and `tail` represent respectively the head of the list (its first element) and its tail (the remainder of the list), and f is an arbitrary function on `sometype`. For illustration, we consider a while statement that contains the following statements (where the dot represents concatenation):

```
while not empty(x)
   {
   ... ... ...
   y:= y.head(x),
   x:= tail(x),
   i:=i-1,
   ... ... ...
   }
```

Application of the first semantic recognizer to the first and second statements produces (after simplification) the following lower bound for $[w]$:

$$V_1 = \{(s,s')|x' = \epsilon \wedge y' = y.x\}$$

where ϵ is the empty sequence. Application of the second recognizer to the second and third line produces (after simplification, using the axiomatization of lists) the following lower bound for $[w]$:

$$V_2 = \{(s,s')|x' = \epsilon \wedge i' = i - length(x)\},$$

where $length(x)$ is the length of x. Taking the join, we find

$$[w] \sqsupseteq \{(s,s')|x' = \epsilon \wedge y' = y.x \wedge i' = i - length(x)\}.$$

4.5 Sample 3-Recognizer

Generally, 3-Recognizers answer the question: what can be infer about the loop function if we know that these three statements get executed the same number of times? We present and illustrate a sample 3-Recognizer in Figure 4. The basic idea of this pattern is to combine the computation of a variable (x) with the use of that variable (in the assignment of a); this is clearly a recurring situation in programs. We briefly illustrate this pattern:

```
w =
while (i<>0) do
   [i:= i-1,
   x:= x-1,
   y:= y+x]
```

The recognizer provides (after ample simplification) the following lower bound for $[w]$:

$$[w] \sqsupseteq \{(s,s')|x \geq i \wedge x' = x - i \wedge$$

$$y' = y + \frac{x(x+1)}{2} - \frac{i(i+1)}{2} \wedge i' = 0\}$$
$$\cup\{(s, s') | x < i \wedge x' = x - i \wedge$$
$$y' = y + \frac{x(x+1)}{2} - \frac{(i-x)(i-x+1)}{2} \wedge i' = 0\}.$$

This function is clearly total, since the domains of the two terms are complementary. It is also deterministic, since the domains of the two terms are disjoint and each term is deterministic. Whence we infer that $[w]$ not only refines this function; it actually equals it.

5 Towards an Automated Tool

5.1 System Architecture

To put this approach into practice, we have resolved to develop a prototype tool that analyzes C++ code and automatically derives its function, or (at least) a lower bound of its function. This tool proceeds by transforming the source code into a function description in three steps:

- *From C++ code to Concurrent Assignments*. For the time being, we assume that the source file, `loop.cpp`, contains variable declarations and a loop, whose body is made up of a sequence of assignments. This steps transforms the loop body from a sequence of assignments to a set of concurrent assignments (in a file that we call `loop.cca`). As of the time of writing, this step has not been developed, but it is fairly straightforward, and can in fact be carried out by a simple compiler generator program (with one semantic rule, interpreting sequence with function composition).
- *From Concurrent Assignments to Lower Bounds*. This step uses theorems 2 and 3 to produce constructive lower bounds. Then it scans its libraries of recognizers and attempts to match them against combinations of concurrent assignments, producing as many lower bounds as it finds matches. These lower bounds are recorded in a file, that we call `loop.mat`.
- *From Lower Bounds to Functional Notation*. File `loop.mat` is submitted to *Mathematica* (©*Wolfram Research*), with a directive to solve the equations in the primed program variables (i.e. the final values of the program variables), and simplify the result. This produces file `loop.bn`.

Figure 5 shows the broad structure of the tool, and we briefly show below a very simple/ simplistic/ naive example to illustrate the transformation process. Our purpose is not to develop a tool that handles such simple minded cases, as we will discuss in the next section; but this example helps us illustrate the process that we envision.

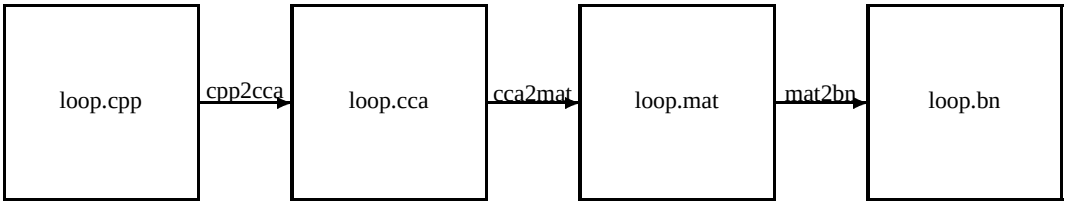

Fig. 5. Broad Architecture of the Tool

5.2 An Illustrative Example

As an illustrative example, we consider the following C++ loop, where all variables are of type integer:

```
loop.cpp:

while x>0
    {x = x-1;
    y = y+2;
    z = z+x+1;
    w = w+2*(y-2);}
```

This program is mapped into the following representation in the concurrent assignment notation:

```
loop.cca

while x>0
    {x = x-1,
    y = y+2,
    z = z+x,
    w = w+2*y}
```

After matching its statements and combinations of statements against our recognizer libraries, we derive the following *Mathematica* input file:

```
loop.mat:
Simplify[Reduce[{
       2*x+y = 2*xP+yP,
       z+(x*(x+1)/2)=
  zP+(xP*(xP+1)/2),
       w-(y*(y-2)/2)=
  wP-(yP*(yP-2)/2),
       xP=0},
       {xP,yP,zP,wP}]]
```

The output produced by *Mathematica* is:

$$\begin{pmatrix} xP = 0 \\ yP = 2x + y \\ wP = w + 2x(x + y - 1) \\ zP = \frac{x+x^2+2z}{2} \end{pmatrix}.$$

We are very encouraged by the fact that Mathematica proves to be very well adapted for this type of manipulations, and we are exploring means to use it for symbolic computations that involve more abstract operators and more complex data types.

5.3 Validation

To validate the calculations performed in the previous section, we wrote the sample program in C++, and wrote an oracle that tests for the function derived above, then tested them on a range of values for variables x, y, z, and w.

```
#include <iostream>
#include <cmath>
using namespace std;

bool testresult;
int xp; int yp; int zp; int wp;
```

```
void loop ();
bool oracle (int x, int y, int z, int w);

int main ()
   {testresult = true;
    t = 2000;

    for (int x = 0; x<10; x++)
{for (int y = 5; y<15; y++)
    {for (int z = 10; z<20; z++)
{for (int w = 15; w<25; w++)
   {xp=x; yp=y; zp=z; wp=w;
    loop();
    testresult = testresult && oracle (x,y,z,w);};};};}
    if (testresult)
       cout << "successful" << endl;
    else
       cout << "unsuccessful" << endl;}

void loop  ()
   {while (xp>0)
       {xp=xp-1; yp=yp+2; zp=zp+xp+1; wp=wp+2*(yp-2);}}

bool oracle (int x, int y, int z, int w)
   {return (xp==0) && (yp==2*x+y) && zp == (x+x*x+2*z)/2 && (wp==w+2*x*(x+y-1));}
```

The program prints `successful`, meaning that the oracle returned `true` for all test values. Then we change the loop by adding an additional variable, t (of type `double`), and adding a statement in the loop body, as shown below.

```
void loop  ()
   {while (xp>0)
       {t = sqrt(t+xp); xp=xp-1; yp=yp+2; zp=zp+xp+1; wp=wp+2*(yp-2);}}
```

For this loop, the space has five variables (x, y, z, w, t). Yet, because we do not have the means to match the assignment to t, the set of lower bounds remains unchanged. Hence our algorithm will produce the same set of equations as before, namely:

$$\begin{pmatrix} xP = 0 \\ yP = 2x + y \\ wP = w + 2x(x + y - 1) \\ zP = \frac{x+x^2+2z}{2} \end{pmatrix}.$$

Whereas these equations define a function on the initial space (x, y, z, w), they define a non-deterministic relation on the new space (x, y, z, w, t). But the equations still hold; when we run the test program with the new version of the loop but with the same oracle, it returns the result `successful`. This is a case where we do not have the function of the loop, but we still have a lower bound of the loop function.

6 Conclusion

6.1 Summary and Assessment

In this paper, we have presented an approach to extracting the function of a while loop. Due to space restrictions, we could not present the approach in detail, hence we focused primarily on discussing its motivations, its premises, and its main tenets. The main idea of this paper is that the function of a loop can be derived in a stepwise fashion, by successive

approximations, where each increment is derived by an arbitrarily partial, arbitrarily localized, analysis of the loop statements (in their concurrent assignment format). This stepwise approach is based on three theorems (theorems 1, 2, 3), which provide lower bounds of the loop function, and a composition rule, which provides means to compose these lower bounds to obtain the function of the loop. The most important theorem, theorem 1, allows us to compose the inductive argument underpinning the loop by identifying reflexive transitive relations that are supersets of the loop body's function. We have illustrated the application of this theorem by showing sample recognizers, which derive reflexive transitive supersets of the loop body by matching statements of the loop body against pre-cataloged patterns. For the sake of combinatorics, it makes sense to build a database of small recognizers (recognizing no more than a few concurrent statements at a time), so that a wide range of loop body structures can be covered with combinations of smaller patterns. Even though we do have recognizers of greater lengths, presented in [23], we are inclined to focus on short recognizers, and see how much mileage (loop coverage) we can achieve with these.

This approach is clearly limited, by the fact that so far, we do not handle *conditional* concurrent assignments. We envision to address this shortcoming by looking into relational mathematics to understand how the transitive closure of union can be derived in a stepwise manner; indeed, concurrent statements of the form:

```
x = f(x,y,z),
y = g(x,y,z),
```

can be interpreted by the intersection of the relations that each line represents. By contrast, statements of the form:

```
t ->  x = f1(x,y,z),
not t ->  x = f2(x,y,z),
```

can be interpreted by the union of the relations that each line represents. Whereas the derivation of loop functions for concurrent assignments uses properties of the transitive closure of an intersection of relations, the derivation of loop functions for *conditional* concurrent assignments is expected to involve properties of the transitive closure of a union of relations.

Also, it is worthwhile to point out that even though most of the examples presented in this paper involve trivial numeric computations (and some trivial list processing), this is not an inherent limitation of the proposed method. The method is as powerful as the recognizers that we use. One way to highlight this power is to derive more general recognizers, and to develop the necessary infrastructure (in the mapping cca2mat) to match them against source code and produce useful lower bounds. This is currently under investigation; the only limitation we see in this step is the ability of *Mathematica* to handle advanced symbolic manipulations.

6.2 Related Work

The derivation of loop functions has not attracted much attention in the past. The only reference we could identify that is specifically geared towards this goal is work by Dunlop and Basili [10]. 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. 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 [24] shows how loop functions can be used to generate loop invariants. Hence in the absence of past work on deriving loop functions, we compare our research to past work on deriving loop invariants, with the qualification that most past work dealt primarily with unary loop invariants, whereas we are concerned here with binary loop invariants. In [11] 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 [6], 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 [26]. In [17, 18] 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 fi xed 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. Less recent work on loop invariants includes work by Cheatham and Townley [4], Karr [16], Cousot and Halwachs [7], and Mili et al [22]. 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 [1, 12].

6.3 Prospects

The work we are presenting in this paper is in its infancy; in this section, we briefly discuss how we envision to expand it.

- *Expanding the Hierarchy of Recognizers*. The capability of the proposed approach is clearly very dependent on the set of recognizers that we have: their number, their generality, their hierarchical structure, etc. We envision to expand the hierarchy of recognizers, at both ends of the tree: at the lower end, to produce more specialized recognizers; and at the higher end, to produce more general recognizers. We envision that our practical experience (instances where we fail to extract a loop function) will drive this growth process.
- *Integrating Conditionals*. Many of the proposed recognizers are based on the premise that some statement gets executed the same number of times as another; this is no longer true once we consider *conditional* concurrent statements. Mathematics must be developed to consider such statements.
- *Dealing with Abstract Data Types*. One of the most hopeful developments in the conduct of our research project is the realization that *Mathematica* can turn the equations that we generate from lower bounds into a functional representation, which indicates the fi nal value of each program variable. This is a very important development, as it means we can focus all our energies on generating lower bounds, and also that we can produce much more complex expressions for lower bounds than if we were responsible for solving the subsequent equations. One of the ways in which we wish to exploit the capability offered by *Mathematica* is to raise the level of abstraction of our recognizers to handle more advanced data types (than numeric types or sequential data types).

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. R. W. Collins, G. H. Walton, A. R. Hevner, and R. C. Linger. The CERT function extraction experiment: Quantifying FX impact on software comprehension and verification. Technical Report CMU/SEI-2005-TN-047, Software Engineering Institute, Carnegie Mellon University, December 2005.
6. M. A. Colon, S. Sankaranarayana, and H. B. Sipna. Linear invariant generation using non linear constraint solving. In *Proceedings, Computer Aided Verifi cation, CAV 2003*, volume 2725 of *Lecture Notes in Computer Science*, pages 420–432. Springer Verlag, 2003.
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. E. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.
10. D. Dunlop and V. R. Basili. A heuristic for deriving loop functions. *IEEE Transactions on Software Engineering*, 10(3):275–285, May 1984.
11. 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.
12. T. Fahringer and B. Scholz. *Advanced Symbolic Analysis for Compilers*. Springer Verlag, Berlin, Germany, 2003.
13. D. Gries. *The Science of programming*. Springer Verlag, 1981.

14. A. R. Hevner, R. C. Linger, R. W. Collins, M. G. Pleszkoch, S. J. Prowell, and G. H. Walton. The impact of function extraction technology on next generation software engineering. Technical Report CMU/SEI-2005-TR-015, Software Engineering Institute, July 2005.
15. C. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576 – 583, Oct. 1969.
16. M. Karr. Affine relationships among variables of a program. *Acta Informatica*, 6:133–151, 1976.
17. L. Kovacs and T. Jebelean. Automated generation of loop invariants by recurrence solving in theorema. In D. Petcu, V. Negru, D. Zaharie, and T. Jebelean, editors, *Proceedings of the 6th International Symposium on Symbolic and Numeric Algorithms for Scientifi c Computing (SYNASC04)*, pages 451–464, Timisoara, Romania, 2004. Mirton Publisher.
18. T. J. L. Kovacs. An algorithm for automated generation of invariants for loops with conditionals. In D. P. et. al., editor, *Proceedings of the Computer-Aided Verifi cation on Information Systems Workshop (CAVIS05), 7th International Symposium on Symbolic and Numeric Algorithms for Scientifi c Computing (SYNASC05)*, pages 16–19, Department of Computer Science, West University of Timisoara, Romania, 2005.
19. R. Linger, H. Mills, and B. Witt. *Structured Programming*. Addison Wesley, 1979.
20. Z. Manna. *A Mathematical Theory of Computation*. McGraw Hill, 1974.
21. A. Mili. Reflexive transitive loop invariants: A basis for computing loop functions. In *First International Workshop on Invariant Generation*, Hagenberg, Austria, June 2007.
22. A. Mili, J. Desharnais, and J. R. Gagne. Strongest invariant functions: Their use in the systematic analysis of while statements. *Acta Informatica*, April 1985.
23. A. Mili, M. Pleszkoch, and R. C. Linger. Towards the automated derivation of loop functions. Technical report, New Jersey Institute of Technology, http://web.njit.edu/˜ mili/loopx.pdf, 2006.
24. H. Mills. The new math of computer programming. *Communications of the ACM*, 18(1), January 1975.
25. C. Morgan. *Programming from Specifi cations*. International Series in Computer Sciences. Prentice Hall, London, UK, 1998.
26. 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.
27. A. Tarski. On the calculus of relations. *The Journal of Symbolic Logic*, 6(3), September 1941.
28. A. Tarski. A lattice-theoretic fixpoint theorem and its applications. *Pacifi c Journal of Mathematics*, (5):285–309, 1955.