A Refinement-Based Framework for Computing Loop Behavior

Ali Mili
College of Computer Science
New Jersey Institute of Technology
Newark, NJ 07102
mili@cis.njit.edu

Abstract

The development, certification and evolution of dependable software requires the ability to analyze software artifacts in all their extensive detail. This, in turn, is contingent upon availability of reliable, certified tools that can rigorously analyze the behavior and properties of software artifacts. One of the most difficult challenges in the development of such a tool is the ability to derive the function of a loop from a static analysis of its source code. In this paper, we discuss the main tenets of our approach to this problem, based on a relation-theoretic refinement calculus, and outline its results, insights, and prospects.

Keywords

Function extraction, loop functions, loop invariants, relational calculus, refinement calculus, computing loop behavior.

1 Introduction: Posing the Problem, Outlining a Solution

1.1 The Function Extraction Project

It is well known that code inspection is one of the most effective techniques for ensuring software product quality. As the size and complexity of software products continue to grow however, it becomes increasingly unrealistic to conduct code inspections without automated support. Also, with software security becoming increasingly important, it is no longer sufficient to ensure that a software product provides the services that are expected from it; we must also ensure that it has no undesirable side-effects (such as malicious code). Finally, the emergence of many third party software development paradigms (such as COTS-based software development or outsourcing) means that we are increasingly dependent on software whose development process we did not control, and whose quality we cannot ascertain. The combination of these conditions places a high premium on having the capability to automatically extract the function of a software artifact in all its details, in all circumstances of use. It is well understood that the problem of computing program behavior is extremely difficult; however the substantial value of such a capability motivates a closer look at what can be done. In this paper, we outline a refinement based approach to the derivation of loop functions by successive approximations, and discuss its strengths, limitations, and prospects.

1.2 Premises of a Stepwise Approach

For the purposes of our discussion in this paper, we represent program functions (that is, the behavior of programs) as *conditional concurrent assignments* (CCA's) or as closed form relational expressions. CCA's express non-procedural definitions of the net behavior of programs and their constituent parts. For example, if we consider the following sequence of assignments on three variables x, y and z (of type integer, say):

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

then their effect can be captured by the following concurrent assignment
$x := x + 1$,
$y := 2 \times (x + 1)$,
$z := z + 2 \times (x + 1)$.
In other words, every concurrent assignment summarizes what happens to a particular variable, and takes

into account the effects of all the sequential statements on that variable. Relational expressions are formed using traditional relational operators such as union, intersection, product, and complement. The focus of our discussion in this paper is the derivation of the function of a while loop, of the form

```
while t do B,
```

where t is a Boolean condition and B is the loop body. The premises that characterize our approach to the problem can be summarized as follows:

- *Closed Form Functions*. We aim to produce a closed form of the loop function; this premise precludes using transitive closure operators, recursive definitions, or existential quantification over the number of iterations. In essence, this mandates the derivation of an inductive argument that derives the loop function from the function of its loop body.

- *Deriving the loop function by successive approximations*. As a divide-and-conquer discipline, the loop function is derived progressively, by accumulating information on the loop behavior as more and more features of the loop are analyzed and captured.

- *Providing substitutes for the loop function*. The loop extraction machinery evolves as more and more programming knowledge and domain knowledge is captured. At any stage in this evolution, we do not merely distinguish between loops that we can handle (whose behavior we can compute) and loops that we cannot handle; rather, we offer a continuum of functional extraction capability, where we can extract the complete function of some loops, most functional attributes of other loops, some functional properties of yet other loops, etc. As the loop extraction machinery evolves, we not only cover more loops, but we also capture more (functional aspects) of each loop. For a given loop, even if we fail to derive its complete function, we may capture part of its functional properties.

- *A refinement based approach*. The ordering and the lattice properties of the refinement ordering are at the core of the divide-and-conquer strategy that we advocate, as well as the strategy of gradually increasing coverage of any loop (until it is fully modeled).

1.3 Related Work

In [10] Dunlop and Basili present a heuristic in which they derive the function of a loop by considering simple expressions of the function under special conditions and attempting to derive the loop function as a generalization of these simpler expressions. The derivation of loop invariants is closely related to the derivation of loop functions since they both aim to derive the inductive infrastructure that underlies the behavior of the loop. Furthermore, a theorem by Basu and Misra [2] shows how loop functions can be used to produce loop invariants and a theorem by Mili et al [18] shows how invariant functions can be used to derive loop functions. Hence the derivation of loop invariants is closely related to the derivation of loop functions, and in the absence of extensive work on deriving loop functions per se, we compare our research to past work on deriving loop invariants. Many researchers in the theorem proving and the program verification communities have lent much attention to the goal of extracting loop invariants [4–8, 11, 14–16, 22, 23]. In the conclusion, we will discuss how these works relate individually to our research; in this section, we discuss in general terms how research on deriving loop invariants differs from research on deriving loop functions along several orthogonal dimensions.

- *Different Goals*. We are trying to derive the function of a loop, whereas the references cited above are geared towards deriving loop invariants; while this distinction is not very profound, we are still dealing with different mathematical objects, that involve different formulas. Perhaps most significantly, the loop invariants are typically used in theorem provers that aim to establish the correctness of a loop program; whereas the functions that we derive are intended for human inspection. This leads to an important distinction in the format of the outputs: whereas loop invariants must be produced in a format that lends itself to subsequent manipulation by a theorem prover, loop functions must be produced in a format that lends itself to human parsing, analysis and inspection.

- *Different Hypotheses*. The function of a loop is dependent exclusively on the loop. By contrast, a loop invariant is typically dependent on a precondition (that defines its basis of induction) and a postcondition (that determines how strong the loop invariant must be to prove the desired correctness property). This is a significant difference, because it means that we look at different sources of information to

derive the loop function and to derive a loop invariant. It also means, incidentally, that a loop has a unique function, but an infinity of loop invariants.

- *Different Scopes*. Loop invariants are meaningful for any iterative program, irrespective of whether it is structured or not. Hence we can talk about the loop invariant of a program that has GoTo's, conditional GoTo's, jumps in and out of the loop body, even when the program does not translate into any structured iterative statement (while, for, repeat until, repeat while, etc). By contrast, loop functions can be derived only for structured loops, at least in the context of this paper.

- *Different Methods*. There is a simple reason why loop functions can be derived only for structured loops whereas loop invariants can be derived even for unstructured loops: Loop invariants are derived by induction on the execution path (the invariant assertion at one point of the execution path is inferred from the invariant assertion at a preceding point of the execution path), whereas loop functions are derived by induction on the control structure (the function of a compound structure is inferred from the functions of its components). Furthermore, the fact that loop invariants are dependent on more information than loop functions (pre/post conditions) means that different methods must be called upon for these two types of tasks: with loop invariants, one usually takes a top down approach, where conditions about program components are inferred from conditions on larger programs (then validated). By contrast, function extraction proceeds strictly in a bottom up manner: functions of compound programs are derived from functions of components.

2 Mathematical Background

For the sake of readability, we keep our discussions throughout this paper at an intuitive level, so that it is not strictly necessary to understand all the details of this section. Those readers who want to follow the detailed mathematics will need to acquaint themselves with the definitions and notations introduced herein.

2.1 Sets and 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 $\widehat{R}$ 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 $dom(\widehat{R})$. 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, $RL = L$). Also, we say that R is *surjective* if and only if $I \subseteq \widehat{R}R$, and that R is *injective* if and only if $R\widehat{R} \subseteq I$. 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 $R = \widehat{R}$.

2.2 A Refinement Calculus

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' (denoted by: $R \sqsupseteq R'$) if and only if*

$$RL \cap R'L \cap (R \cup R') = R'.$$

This definition is consistent, modulo differences of notation, with traditional definitions of refinement [13, 21]. Intuitively, this property is the relational equivalence of the property that provides that a specification refines another if its precondition is weaker and its post-condition is stronger. We admit without proof that the following propositions hold, modulo traditional definitions of total correctness [9, 13, 17]:

- 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'.

In [3] we have studied the lattice properties of this ordering, and found the following results:

- Any two relations R and R', which satisfy the following condition

  $$RL \cap R'L = (R \cap R')L$$

  have a *join* (i.e. least upper bound), which is defined by:

  $$R \sqcup R' = R \cap \overline{R'L} \cup R' \cap \overline{RL} \cup R \cap R'.$$

- Any two relations R and R' have a *meet* (greatest lower bound), which we denote by $R \sqcap R'$, and define by

  $$R \sqcap R' = RL \cap R'L \cap (R \cup R').$$

The lattice of refinement admits a *universal lower bound*, which is the empty relation. It admits no *universal upper bound*; maximal elements of this lattice are total deterministic relations. The expressions of join and meet are complex because our relational specifications can be arbitrarily partial (are not necessarily total), and arbitrarily non-deterministic (are not necessarily functions). The outline of Figure 1 shows the overall structure of the lattice of refinement.

3 Basis of a Stepwise Approach

Several approaches are possible in deriving the function of a loop. The core idea of the approach that we are advocating in this paper is to derive the function by considering a few statements at a time, thereby obviating the need to face the size and complexity of the loop all at once. In this section we discuss, in turn, how to derive partial claims about the function of the loop, then how to combine these partial claims to obtain a (more) complete description of the loop function.

3.1 Deriving Partial Claims

We distinguish between two aspects of the question of deriving / discovering partial functional details about the function of a loop: first, how do we discover such partial functional details; second, how do we represent them. We discuss the first question in section 4; and we discuss the second question in this section. We let w be a while statement on space S written as:

```
while t do B
```

and we let $[w]$ be the function that this while statement defines on space S. All claims about the function $[w]$ of the while statement w take the form of a refinement statement:

$$[w] \sqsupseteq T$$

for some specification T; the higher the specification T in the refinement ordering (see Figure 1) the stronger the statement. Borrowing terminology from lattice theory, we refer to T as being a *lower bound* for $[w]$. The following theorems, due to [20], provide lower bounds for $[w]$, and form the basis for our stepwise approach; they all assume that the loop terminates for all initial states. In [20], we find that this assumption does not affect generality, in the following sense: all initial atstes, all final states and all initial states in the execution of a loop w fall within the domain of $[w]$. Hence if we redefine the state space of the loop as its domain, we do not exclude any state of interest; in practice, this means that the first step we must take before deriving the function of a loop is to find its domain (by no means a straightforward task), then we redefine its space as the domain. We provide below the three theorems, due to [20], that we use to derive lower bounds of the loop function; we do not provide proofs for these theorems, but briefly comment on what they mean.

Theorem 1 *We consider a while loop w on space S, defined by w =* `while t do B`. *If R is a reflexive transitive relation such that*

$$I(t) \circ [B] \subseteq R \wedge R \circ I(\neg t) \circ L = L,$$

then

$$[w] \sqsupseteq R \circ I(\neg t).$$

We comment on the conditions of this theorem:

- *R must be a superset of $[B]$.* Given that the function of the loop body is defined by a set of concurrent assignments, a superset of this function can be derived by inspecting an arbitrary (arbitrarily small) subset of these concurrent assignments. This is a crucial **separation of concerns** attribute.

- *R must be reflexive and transitive*. Extracting the function of a loop consists essentially in **identifying the inductive argument that underlies the loop**[1]. We submit that the derivation of a reflexive and transitive superset of the loop body captures the inductive argument of the loop, in the following sense: reflexivity serves the basis of induction, and transitivity serves the inductive step of the claim that the lower bound derived from R is refined by $[w]$.

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

$$[w] \sqsupseteq T$$

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

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
```

[1]Unless the loop is incorrect, this consists in recovering the inductive argument that the programmer used in building the loop in the first place.

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*

$$[w] \sqsupseteq T$$

where T is defined as:

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

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 $T = L \circ I(\neg t)$, which merely expresses that the final state satisfies $\neg t$ (which is redundant with the other theorems).

3.2 Composing Partial Claims

By applying theorem 1 repeatedly (for a variety of combinations of concurrent assignments), and by applying theorems 2 and 3 as needed, we derive a set of lower bounds for $[w]$, which we denote by T_1, T_2, ... T_k. Then two questions arise: first, how do we compose T_1, T_2, ... T_k? second, how do we know that we have enough lower bounds to derive the function of the loop? We answer these questions below:

- *Combining Lower Bounds*. Using a simple identity from lattice theory, we find that if $[w]$ refines lower bounds T_1, T_2, ... T_k, then it refines their join (least upper bound). Hence we write:

  $$[w] \sqsupseteq T_1 \sqcup T_2 \sqcup .. \sqcup T_k.$$

  We have seen in section 2.2 that the join of specifications is not always defined; but we know from a theorem due to [3] that a set of terms have a join if and only if they have an upper bound (in this case, T_1, T_2, ... T_k clearly do have an upper bound, which is $[w]$). See Figure 1. The sequence

  $$T_1, T_1 \sqcup T_2, T_1 \sqcup T_2 \sqcup T_3, ..., T_1 \sqcup T_2 \sqcup ... \sqcup T_k$$

  represents successive approximations of the loop function.

- *Convergence Condition*. As we recall from section 2.2, maximal elements of the lattice of refinement

are relations that are total and deterministic. An element T is maximal if and only if

$$\forall W : W \sqsupseteq T \Rightarrow W = T.$$

Whence we find that whenever a set of lower bounds is such that their join

$$T_1 \sqcup T_2 \sqcup ... \sqcup T_k$$

is total and deterministic, we can infer (by the modus ponens):

$$[w] = T_1 \sqcup T_2 \sqcup ... \sqcup T_k.$$

In practice, it means that for a given while statement w, we extract all the lower bounds we can and take their join. If their join is total and deterministic, then it is the function of the loop; if not, then it is as much information as we can gather about the loop with current capabilities (this will be clarified further in section 4).

4 Extracting Partial Functionality

4.1 Sketch of an Algorithm

In section 3.1, we have seen that we can derive lower bounds for the loop function using theorems 1, 2 and 3. While the latter two theorems are constructive, in the sense that they produce an explicit expression of a lower bound for $[w]$, theorem 1 requires creativity: we must produce a relation R, then check that it satisfies the conditions of the theorem; if it does, then we use R to derive the corresponding lower bound T. To help derive such lower bounds, we provide program patterns, called *recognizers*, which identify specific code patterns and map them directly into lower bounds (skipping the intermediary step of generating R). A recognizer is characterized by its state space, the pattern of 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 CCA's, 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. In the sequel, we present sample recognizers, then briefly discuss how the set of recognizers is structured for optimal operation.

4.2 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 $[w]$ refines the following specification:

$$T = \{(s, s') | x \bmod c = x' \bmod c \wedge y' \leq 0\}.$$

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.

4.3 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:

```
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 $[w]$:

$$T_1 = \{(s, s') | x' = \epsilon \wedge y' = y.x\}$$

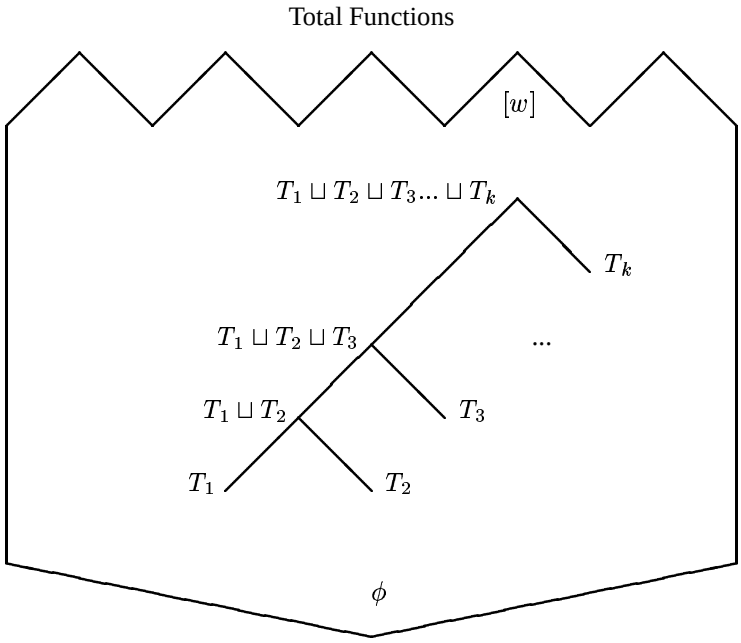

Figure 1. Successive Approximations of the Loop Function

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

Figure 2. Sample 1-Recognizer

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

Figure 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	$T =$ $\{(s,s') \mid f^i(x) = f^{i'}(x') \wedge$ $a + \Sigma_{k=1}^{i} f^k(x) = a' + \Sigma_{k=1}^{i'} f^k(x') \wedge \neg t(s')\}$

Figure 4. Sample 3-Recognizer

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 $[w]$:

$$T_2 = \{(s, s') | x' = \epsilon \wedge i' = i - length(x)\},$$

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

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

4.4 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. This pattern can be generalized in many ways (generalizing the '+', the increment of i, the direction of the increment, etc) but we keep it simple here. 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.

4.5 A Hierarchy of Recognizers

We view the set of semantic recognizers not as an unstructured monolith, but rather as a hierarchical structure ordered by generality. Also we envision that an algorithm that attempts to recognize patterns in the source code to derive lower bounds of the loop function attempts first to match lower level patterns (for any length: 1, 2, or 3), and climbs up the tree only if lower level patterns do not produce a match with the source code. There exists a tension between generality/ usefulness and genericity/ usability that arises in defining these patterns. To illustrate the contrast between generality (usefulness) and genericity (usability), we consider the following example: We know from [20] that if the loop body contains two statements such as

SR: `i:= i-1, x:= x+i`

then the loop refines the following relation (where t is the loop condition)

$$T = \{(s, s') | x + \frac{i(i+1)}{2} = x' + \frac{i'(i'+1)}{2} \wedge \neg t(s')\}.$$

Because these two statements are too specific, we may want to generalize them into the following form:

SR': `i:=i-c, x:=x`$\oplus$`i`.

Now the step by which we decrement i is arbitrary, and the operation by which we compose x and i is also arbitrary; whence we have obtained a more general pattern, that is more widely applicable. But this additional generality comes at a cost in terms of usability, since now the lower bound of the loop function has the following form (provided $\oplus$ is associative):

$$T' = \{(s, s') | x \oplus \bigoplus_{k=1}^{i \div c} k \times c = x' \oplus \bigoplus_{k=1}^{i' \div c} k \times c\}.$$

SR' is more general than SR, hence more widely applicable, but the relation it produces is less tractable (less usable) since now we must parse it with the instantiated operator, simplify it by means of algebraic properties of the instantiated operator, use identities that are specific to the instantiated operator, etc. By contrast, relation T is readily usable as it is.

5 Conclusion

5.1 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. 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-catalogued patterns. For the sake of combinatorics, it makes sense to build a database of small recognizers (recognizing no more than three concurrent statements at a time), so that a wide range of loop body structures can be covered with combinations of smaller patterns.

5.2 Relation to Other Work

In section 1.3 we have discussed in general terms how research on loop invariants is similar to our research on functional extraction, and how it differs from it. In this section we briefly mention some samples of current research on loop invariants, and characterize their approach. In [11] Ernst et al. discuss a system for dynamic detection of likely invariants; this system, called Daikon, runs candidate programs and observes their behaviour 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 paramerize with relevant safety properties. Safety properties are formulated by invariants (including loop invariants), which are inferred by propagation through the code. In [6], Colon et al. consider loop invariants of numeric programs as linear expressions and derive the coefficients of the linear expressions by solving a set of linear equations; they extend this work to non linear expressions in [22]. In [15, 16] 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 [4] 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. Less recent work on loop invariants includes work by Cheatham and Townley [5], Karr [14], Cousot and Halwachs [7], and Mili et al [18]. 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].

5.3 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.

- *Integrating ADT axiomatizations*. Most of the current recognizers are focused on the control structure of the loop body, and are assuming numeric data types, whose axiomatization is implicit. More advanced data types require that we integrate the axiomtization of the relevant ADT's into the functional extraction machinery.

- *Integrating Loop Invariant Methods*. Many of the methods we discussed in section 5.2 provide means to derive loop invariants automatically, from a runtime analysis of the loop structure. By contrast, the method discussed here is based on a set of patterns

that are derived off-line and matched against statements of the loop body. Dynamic procedures must be developed to extract loop invariants (that play the role of relation R in theorem 1) even in cases where no pre-stored patterns provide a match.

- *Managing Recognizer Libraries*. Many of the tradeoffs that arise in defining, storing and managing recognizers are reminiscent of those that arise in managing reusable software assets. The body of knowledge and experience gained in the management of software asset libraries may prove valuable in managing recognizer libraries [19].

Acknowledgment

The author wishes to thank Dr Richard C Linger, from the Software Engineering Institute, for his extensive feedback on an earlier version of this paper.

References

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

[2] S. Basu and J. Misra. Proving loop programs. *IEEE TRansactions on Software Engineering*, 1(1):76–86, March 1975.

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

[4] 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.

[5] 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.

[6] 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.

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

[15] 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 Scientific Computing (SYNASC04)*, pages 451–464, Timisoara, Romania, 2004. Mirton Publisher.

[16] 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 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.

[17] Z. Manna. *A Mathematical Theory of Computation*. McGraw Hill, 1974.

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

[19] A. Mili, R. Mili, and R. Mittermeir. A survey of software reuse libraries. *Annals of Software Engineering*, 1998.

[20] 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.

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

[22] 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.

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