Toward the Automated Derivation of Loop Functions

Ali Mili
College of Computer Science
New Jersey Institute of Technology
Newark NJ 07102-1982
mili@cis.njit.edu

Mark Pleszkoch and Richard C. Linger
Software Engineering Institute
Carnegie Mellon University
Pittsburgh PA 15213-3890
{rlinger,mpleszko}@sei.cmu.edu

January 29, 2007

Abstract

The criticality of modern software applications, the pervasiveness of malicious code concerns, the emergence of third party software development, and the preponderance of program inspection as a quality assurance method, all place a great premium on the ability to analyze programs and derive their function in all circumstances of use and all its functional detail. One of the most challenging tasks in this endeavor is, clearly, the derivation of loop functions. In this paper, we outline the premises of our approach to this problem, present and illustrate some preliminary results, and sketch some tentative directions of research.

Keywords

Functional extraction, loop function, sub-goal induction theorem, Mills' theorem.

1 Introduction: Motivation and Premises

1.1 Motivation

The ever increasing size and complexity of software systems [26] places extraordinary demands on human ability to understand and analyze programs. Yet, paradoxically, the need to understand programs, i.e. determine what a program

does in all circumstances of use, remains as pressing as ever, for a multitude of reasons, including:

- 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 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? To answer this type of question, we need means to capture the function of a 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 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. A functional extraction prototype that computes the behavior of programs written in a small subset of Java was used in a rigorous experiment to quantify the value of automated behavior calculation. A control group using traditional program reading and inspection and an experimental group using the FX prototype were each given the same small programs and questions to answer. Results showed the experimental group provided four times more correct answers to the questions in one-fourth of the time required by the control group [6]. Another study conducted with a major corporation investigated the potential impact of FX technology on the software engineering life cycle [15].

Whereas the derivation of program functions for sequential code segments and for if-then-elses is fairly straightforward, the derivation of loop functions is rather difficult. The automated derivation of loop functions for statements of the form *while t do B* is the subject of this research. For the purposes of this study, we assume that B is represented by a set of concurrent assignments. Concurrent assignments 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 assignments

$x := x + 1,$

$y := 2 \times (x + 1),$

$z := z + 2 \times (x + 1).$

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.

1.2 Premises

The premises that characterize our approach to the derivation of loop functions 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 means that we must bridge the inductive gap between the function of the loop body (which describes what happens in a single iteration) and the function of the loop (which describes what function the whole loop computes).

- *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. This is a crucial feature of our approach, as it makes it possible to derive the function of arbitrarily large loops by analyzing small segments of their source code at a time.

- *Providing substitutes for the loop function*. The process outlined in the previous item (of stepwise accumulation of functional attributes) may terminate before we have obtained a function (this happens if some features of the loop are beyond the reach of our current loop extraction capability); in that case, we have partial functional information about the loop, but not a complete description of the loop's functionality. The loop extraction machinery evolves as more and more programming knowledge (pertaining to programming language semantics)

and domain knowledge (pertaining to domain-specific abstractions) is captured. At any stage in this evolution, we do not merely distinguish between loops that we can handle (whose behavior we can compute in full) 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.

- *A refinement based approach.* The ordering properties 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). The refinement ordering gives us a framework in which we can cast our arguments and our algorithms.

1.3 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 [11]. 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 [23] shows how loop functions can be used to produce loop invariants. Hence in the absence of past work on deriving loop functions, 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, 5, 7, 8, 16–18, 27, 28]. We first 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 most of 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 loop

functions that we derive are intended for human 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 that a loop has a unique function, but potentially an infinity of loop invariants (one for each precondition/ postcondition pair).

- *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 work.

- *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 typically 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.

- *Different Forms*. In the process of deriving loop functions, we come across expressions which can be interpreted as loop invariants, but they are very different from loop invariants that are derived/ discussed in the references we cited above, in at least two aspects: The first (trivial) difference is that traditional loop invariants are unary (in the current state) whereas the loop invariants we derive are binary, referring to a current state and a past state;

the second, more meaningful, difference is that unary loop invariants capture properties of the current state with respect to an initial state, whereas binary loop invariants capture properties between a past state and the current state, where both states are arbitrary states in the iteration sequence. Interestingly, in most cases the binary loop invariants are represented by a symmetric relation, where the roles of the current state and the past state are interchangeable (in other words, the loop invariant reflects that the states are a number of iterations apart, but does not tell which state precedes which). This will be further clarified in the sequel (see comment at the end of section 4.3).

- *Different Data Types*. Most work we have seen on deriving loop invariants deals mostly with numeric data types (most often integer/ natural variables). At the level of abstraction of our work, no such a restriction is necessary. We will briefly cite examples where non-numeric data types are used.
- *Different Levels of Abstraction*. As a result of being restricted to numeric data types, most work we have seen on deriving loop invariants is bound to the level of abstraction of numeric operators. By contrast, our approach can operate at arbitrary levels of abstraction, as we will illustrate in the sequel.
- *Different Means*. Whereas work on extracting loop invariants uses logical manipulations, we use algebraic tools that stem from a relational refinement calculus. Using different means often yields complementary insights.

Now we briefly review and characterize some work on deriving loop invariants. In [12] 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 [9], 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 [7], 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 [27]. 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 [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 [16], Cousot and Halwachs [8], 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, 13].

2 Mathematical Background

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 $R^\frown$, 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 total function (total deterministic relation) F, we find that the nucleus of F is an equivalence relation; the equivalence classes of S modulo the nucleus of F are called the *level sets* of F; each equivalence class represents a set of elements of S that have the same image by F.

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* [29, 30], which provides that for all relation R, $LRL = L$ if and only if R is non-empty. 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 [14, 24]. We abbreviate this property by $R \sqsupseteq R'$ or $R' \sqsubseteq R$. We admit that, modulo traditional definitions of total correctness [10, 14, 19], 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 [3] 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. 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 1 *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:

$$R \sqcap R' = R \cup R'.$$

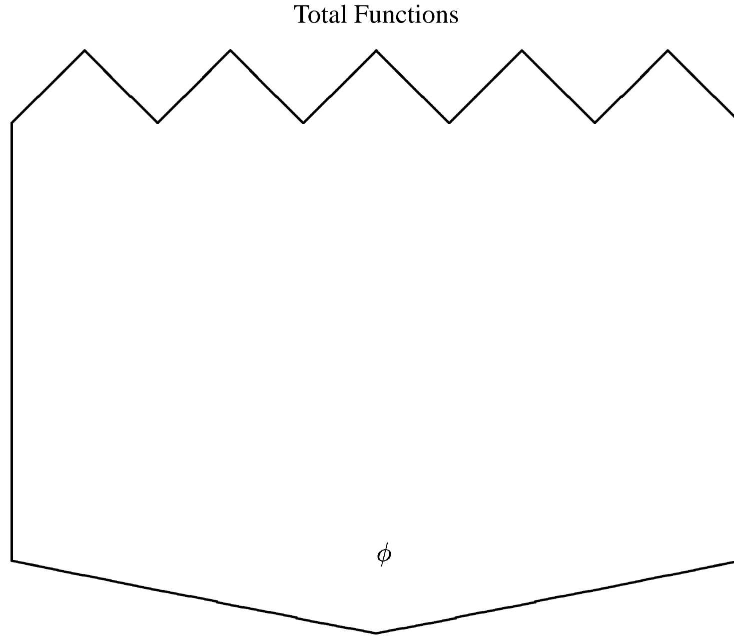

Figure 1: Lattice Structure of Refinement

3 Tenets of a Stepwise Approach

We consider a program P on some variables x_1, x_2, ... x_n. We let S be the space defined by all the values that the aggregate of variables may take. The function of program P is denoted by $[P]$ and defined by

$$[P] = \{(s, s')| \text{ if } p \text{ starts execution on } s \text{ then it terminates in state } s'\}.$$

From this definition, it stems that *dom*$([P])$ can be interpreted as

$$dom([P]) = \{(s, s')| \text{ if } P \text{ starts execution on state } s \text{ then it terminates}\}.$$

We submit two fundamental theorems about loops, which we will use subsequently.

Theorem 1 Sub-goal Induction Theorem. *Given a while loop* $w =$ `while t do B` *on space* S *and a total relation* R *on the same space* S *(such that* $dom(R) = S$*),* w *is correct with respect to* R *if:*

- $dom([w]) = S$.
- $I(\neg t) \subseteq R$.

- $I(t) \circ [B] \circ R \subseteq R$.

This theorem is due to Morris and Wegbreit [25].

Theorem 2 Mills Theorem. *Given a while loop* $w =$ `while t do B` *on space* S *that terminates for all states in* S*, and a function* F *on the same space* S*, then*

$$[w] = F$$

if and only if:

- $dom(F) = S$.
- $I(\neg t) \circ F = I(\neg t)$.
- $I(t) \circ [B] \circ F = I(t) \circ F$.

This theorem is due to H.D. Mills [23]. Even though it was derived independently from (and prior to) the Sub-goal Induction Theorem, it could be considered as a special case of it (the case where the specification at hand, R, is deterministic).

Because these two theorems assume that $[w]$ is total, we are interested in restricting our study to loops that meet this condition; the proposition below provides that this assumption causes no loss of generality.

Proposition 2 *We consider a while loop* w *on space* S*. The hypothesis* $dom([w]) = S$ *does not cause a loss of generality.*

Proof. Initial states are by definition in $dom([w])$, since they are states for which execution of the loop terminates. Final states are also in $dom([w])$, since they satisfy condition $\neg t$, hence the execution of the loop on such states terminates immediately. Intermediate states are also in $dom([w])$, since if they weren't, the loop would not terminate for them, whence it would not terminate for the initial states from which they come. **qed**

Since initial states, intermediate states, and final states are all in $dom([w])$, we can let S be $dom([w])$ without loss of generality, as then all the states of interest are within S. This choice of state space makes the while statement's function vacuously total. In the sequel, we implicitly assume this condition throughout, unless otherwise specified.

What this means, in practice, is that whenever we are given a while loop on some space S', we let S be the subset of S' that represents the domain of $[w]$, and we discuss the loop extraction of w on space S. By making this assumption, we are not presuming that the derivation of the domain of $[w]$ is easy in practice; it is often very difficult, and we are separately exploring means to derive it. But our subsequent discussion holds only for cases where $[w]$ is total, or, conversely, where the space is restricted to the domain of $[w]$.

The following theorem gives an explicit expression for the function of a while loop.

Theorem 3 *Given a while statement of the form* $w =$ `while t do B`*. If* w *terminates for all the states in* S*, then*

$$[w] = (I(t) \circ [B])^* I(\neg t).$$

Proof. We use Mills' theorem. To this effect, we let F be the function on S defined by

$$F = (I(t) \circ [B])^* I(\neg t)$$

and we check in turn the three conditions of this theorem.

First condition: $dom(F) = S$. We assume that there exists an element of s of S that is outside $dom(F)$. Because this element is in S, and because $[w]$ is total, execution of w on s terminates in a state s'. We infer that there exists a natural number n such that

$$(s, s') \in (I(t) \circ [B])^n I(\neg t),$$

(n is the number of iterations it takes to produce s' from s). Whence we infer

$$s \in dom((I(t) \circ [B])^n I(\neg t)),$$

from which we infer, in turn,

$$s \in dom((I(t) \circ [B])^* I(\neg t)),$$

(by definition of the reflexive transitive closure), from which we infer (by substitution),

$$s \in dom(F),$$

which contradicts the assumption that s is outside the domain of F.

Second condition:

$$I(\neg t) \circ (I(t) \circ [B])^* I(\neg t)$$

$=$ $\{$ substitution $\}$

$$I(\neg t) \circ ((I(t) \circ [B])^+ \circ I(\neg t) \cup I(\neg t))$$

$=$ $\{$ distributivity $\}$

$$I(\neg t) \circ ((I(t) \circ [B])^+ \circ I(\neg t)) \cup I(\neg t)^2$$

$=$ $\{$ simplification $\}$

$$I(\neg t)^2$$

$=$ $\{$ relational identity $\}$

$$I(\neg t).$$

Third Condition:

$$I(t) \circ [B] \circ [w]$$

$=$ $\{$ Substitution $\}$

$$I(t) \circ [B] \circ (I(t) \circ [B])^* \circ I(\neg t)$$

$=$ $\{$ Relational Identity $\}$

$$(I(t) \circ [B])^+ \circ I(\neg t)$$

$=$ $\{$ Prerestricting a relation to a superset of its domain $\}$

$$I(t) \circ (I(t) \circ [B])^+ \circ I(\neg t)$$

$=$ $\{$ Adding a null term $\}$

$$I(t) \circ (I(t) \circ [B])^+ \circ I(\neg t) \cup I(t) \circ I(\neg t)$$

$=$ $\{$ Left factoring $\}$

$$I(t) \circ ((I(t) \circ [B])^+ \circ I(\neg t) \cup I(\neg t))$$

$=$ $\{$ Right factoring $\}$

$I(t) \circ ((I(t) \circ [B])^+ \cup I) \circ I(\neg t)$

= { Definition of Reflexive Transitive Closure }

$I(t) \circ ((I(t) \circ [B])^*) \circ I(\neg t)$

= { Substitution }

$I(t) \circ [w]$.

qed

This theorem gives an explicit expression for the function of the while loop, but it is of little help, since in general we do not know how to compute the transitive closure of a relation. The purpose of the following theorem is to allow us to obviate the need to compute the transitive closure, by means of a separation of concerns strategy that attains the function of the loop by collecting arbitrarily weak approximations of it.

Theorem 4 *We consider a while loop w on space S, defined by $w =$* `while t do B.` *If R is a total transitive relation such that*

$$I(t) \circ [B] \subseteq R$$

and

$$R \circ I(\neg t) \circ L = L,$$

then

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

Proof. We let T be defined as

$$T = (R \cup I) \circ I(\neg t).$$

and we note that T is total, since

TL

= { substitution }

$(R \cup I) \circ I(\neg t) \circ L$

$\supseteq$ { monotonicity }

$R \circ I(\neg t) \circ L$

$=$ { hypothesis of the theorem }

$L.$

We use definition 1, which calls for computing/ analyzing the following expression:

$[w]L \cap TL \cap ([w] \cup T)$

$=$ { Because $[W]$ is total }

$TL \cap ([w] \cup T)$

$=$ { Because T is total, as shown above }

$([w] \cup T)$

$=$ { substituting $[w]$ and T }

$(I(t) \circ [B])^* \circ I(\neg t) \cup (R \cup I) \circ I(\neg t)$

$=$ { factoring, associativity }

$((I(t) \circ [B])^* \cup R \cup I) \circ I(\neg t)$

$=$ { substitution }

$((I(t) \circ [B])^+ \cup I \cup R \cup I) \circ I(\neg t)$

$=$ { simplification }

$((I(t) \circ [B])^+ \cup R \cup I) \circ I(\neg t)$

$=$ { Because R is transitive }

$((I(t) \circ [B])^+ \cup R^+ \cup I) \circ I(\neg t)$

$=$ { monotonicity of transitive closure, and hypothesis }

$(R^+ \cup I) \circ I(\neg t)$

$=$ { because R is transitive }

$(R \cup I) \circ I(\neg t)$

$=$ { substitution }

T.

qed

The interest of this theorem stems from the observation that whenever we find a relation R that satisfies the conditions of the theorem, we can derive from it a specification T that $[w]$ refines. Such a specification T gives us some (typically partial) information on the loop, and can be used in a stepwise derivation of the loop function. Theorem 3 gives an explicit expression of the loop function, in terms of the transitive closure of the loop body; as such, it is of little use in practice because we cannot generally derive the transitive closure of a known function. This theorem provides us with a compromise: Rather than ask us to compute the transitive closure of the loop body, it merely asks that we find a transitive superset of the loop body; in exchange, the theorem does not produce the loop function but produces instead a lower bound of the loop function. If we find enough lower bounds of the loop function, we may be able to derive it by taking the join of all the lower bounds.

This is of course contingent on the assumption that is it easier to derive a transitive superset of the loop body than it is to derive its transitive closure. This assumption is borne out because the transitive closure of a relation is the *smallest* transitive superset thereof. What theorem 4 does is to lift the requirement of being the smallest, and allows us to select arbitrarily large transitive supersets of the loop body. In order to derive the smallest superset of a loop body represented by concurrent assignments, we need to look at all the assignments; but in order to derive an arbitrary transitive superset, we may look at any subset of the concurrent assignments.

Another interpretation of this theorem is that it substitutes the (difficult) task of computing the transitive closure of an intersection of relations by the tasks of computing the transitive closures of the terms of the intersection.

Corollary 1 *If R is a transitive relation such that*

$$[B] \subseteq R$$

and

$$R \circ I(\neg t) \circ L = L,$$

and w is the while loop defined by `while t do B` *then*

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

This corollary stems readily from theorem 4 since $I(t) \circ [B]$ is a subset of $[B]$. The interest of this corollary: we can separate the analysis of the loop body from the analysis of the loop condition; we only look at $[B]$ to derive transitive supersets of it. And the condition we test on R once it is derived involves t only very marginally.

Corollary 2 *If R is a reflexive transitive relation such that*

$$[B] \subseteq R$$

and

$$R \circ I(\neg t) \circ L = L,$$

and w is the while loop defined by `while t do B` *then*

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

This stems readily from the property that for such relations $(R \cup I) = R$ and the property that a reflexive relation is necessarily total. This corollary is applicable, of course, to equivalence relations, which will arise in many loop examples.

This corollary allows us to discuss an important attribute of the proposed approach. In the introduction, we had mentioned that our purpose, like the purpose of loop invariant generation, is to discover the inductive argument that underlies the operation of the loop. We argue that generating a reflexive transitive superset of the loop body is at the core of the inductive analysis of the loop, in the following sense:

- Reflexivity serves as the basis of induction, and

- Transitivity serves as the inductive step,

of a proof by induction (on the number of iterations) to the effect that $[w]$ refines $R \circ I(\neg t)$. While $[B]$ captures the effect of one execution of the loop body, R captures the effect of zero (reflexivity) or more (transitivity) executions of the loop body.

Theorem 5 *Let w be a while loop defined by* `while t do B`. *If $t \not\equiv$* **false** *then*

$$[w] \sqsupseteq T$$

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

Proof. We use the *sub-goal induction theorem*. To this effect, we must prove the following premises:

- T is total.
- $[w]$ is total.
- $I(\neg t) \subseteq T$.
- $I(t) \circ [B] \circ T \subseteq T$.

The second premise has been assumed to hold throughout, by virtue of Proposition 2. The third premise stems readily from inspecting T. To prove the first premise, we introduce a simple lemma, to the effect that

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

Tarski's identity provides that an expression of the form LQL equals L for all Q, except if $Q = \phi$. Hence to prove our lemma, all we need to show is that

$$I(t) \circ [B] \circ I(\neg t)$$

is not empty. If this relation were empty, we would infer that statement B cannot map a state that satisfies t into a state that satisfies $\neg t$. We know by definition that $rng([w]) = \{s|\neg t(s)\}$; and we know by hypothesis that $t \not\equiv$ **false** . Hence $rng([w]) \neq S$. Let s be an element of $\overline{rng([w])}$. Because state s is outside $rng([w])$, it satisfies predicate t; whence application of the loop on s will execute B; because we are assuming that

$$I(t) \circ [B] \circ I(\neg t)$$

is empty, application of B on s will necessarily produce a state s' that satisfies t; for the same reason as above, application of B on s' will produce a state that satisfies t, etc... Hence we infer that application of the loop to s does not terminate, which is in contradiction with the hypothesis that s is in $dom([w])$. Whence we conclude that

$$I(t) \circ [B] \circ I(\neg t) \neq \phi,$$

therefore

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

To prove that T is total, we compute TL:

TL

$=$ { substitution }

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

$=$ { associativity }

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

$=$ { lemma above }

$I(t) \circ L \cup I(\neg t) \circ L$

$=$ { factorization }

$(I(t) \cup I(\neg t)) \circ L$

$=$ { simplification }

$I \circ L$

$=$ { identity }

$L.$

The proof of the fourth premise is straightforward:

$I(t) \circ [B] \circ T$

$=$ { substitution }

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

$\subseteq$ { set theory }

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

$=$ { associativity }

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

$\subseteq$ { monotonicity }

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

$\subseteq$ { by inspection }

$T.$

qed

We need this theorem because in many cases it is not sufficient to know that the initial state of the loop satisfies t and the final state satisfies $\neg t$. It is also necessary to know that the final state s' is the first state that does not satisfy t in the sequence of applications of $[B]$. In other words, there exists an antecedent of s' by $[B]$ that satisfies t. Note that this applies only if the loop iterates at least once, whence the condition

$$t \not\equiv \mathbf{false}\ ,$$

which excludes the pathological case where the loop never iterates (when the loop function is only the identity). As an example, consider the simple loop

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

where i is an integer variable. If we characterize the final states of this loop by simply saying that they satisfy $\neg t$, then all we know about the final value of i is that it is less than or equal to 1. But in fact we know more than that: we know that if the loop starts with a value of i greater than 1, it terminates with a value of 1. The theorem above allows us to make such a claim by characterizing the final state with two properties: 1) It satisfies $\neg t$; 2) its antecedent by $[B]$

satisfies t. Intuitively, this theorem is useful whenever the loop condition is of the type $(i > 1)$ rather than the type $(i \neq 1)$.

4 Divide and Conquer Strategies

4.1 A Numeric Analogy

Before we present our refinement based approach to the extraction of loop functions, we briefly present an illustrative example, using a lattice-like structure that has the same properties as the structure of the refinement ordering. 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.

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.

4.2 Climbing the Lattice of Refinement

The approach we advocate in this paper is to derive the function of the while loop by accumulating refinement claims of the form

$$[w] \sqsupseteq T_i$$

for as many lower bounds (T_i) as possible; all the theorems and corollaries of section 3 are intended to produce such lower bounds. The usefulness of these theorems and corollaries is that they allow us to compose the function of the loop in a stepwise manner by looking at arbitrarily small pieces of the loop, and in fact by separating the loop body from the loop condition. The questions that arises then are:

- **First, how do we combine statements of the form**

  $$[w] \sqsupseteq T_i$$

  into a cumulative claim about $[w]$**?** The answer is quite simple, and stems from the lattice properties of the refinement ordering [3]. If we have a finite set of specifications T_1, T_2, ... T_k that are all refined by $[w]$, then we can infer that

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

  There is no question of whether the join is defined, since $[w]$ is an upper bound for the T_i's, and we know from [3] that a join (lowest upper bound) exists if and only if an upper bound exists.

- **How do we know when to stop**? A straightforward answer to this question is: we stop when the join of all the relations we have derived produces a total deterministic relation. A more nuanced answer is: we stop when we run out of lower bounds, or when the join of all the lower bounds we have satisfies our goals. We often find examples where deriving a function (rather than a sufficiently informative / deterministic relation) may be unrealistic, prohibitively expensive, or unnecessary.

In practice, it may be more economical not to apply theorem 4 and its corollaries to each transitive relation R_i to derive relation T_i, but rather to collect all the relations R_i, take their intersection, say R, then apply the theorem (or its corollary) to R. This is possible because the finite intersection of transitive relations is transitive, and the finite intersection of reflexive relations is reflexive.

Proposition 3 *Under the conditions of theorem 4, the join of T specifications is defined and equals their intersection. Furthermore, the intersection of all the T specifications is total.*

Proof. To fix our ideas, we write the proof for two specifications T_1 and T_2, and let the reader infer its generalization to an arbitrary number of such specifications. From the claims

$$[w] \sqsupseteq T_1$$

$$[w] \sqsupseteq T_2$$

we infer that T_1 and T_2 have an upper bound, namely $[w]$, whence we infer (from a result due to [3]) that T_1 and T_2 have a least upper bound. According to theorem 4, T_1 and T_2 are total, hence have the same domain. By virtue of Proposition 1 their join (least upper bound) is their intersection.

Also, according to [3], the existence of a join is equivalent to the following condition

$$T_1 L \cap T_2 L = (T_1 \cap T_2) \circ L.$$

Because T_1 and T_2 are total, the left hand side equals L, whence so is/ does the right hand side. **qed**

Proposition 4 *If $[w] \sqsupseteq T$ and T is total and deterministic then $[w] = T$.*

This Proposition stems readily from the structure of the lattice of refinement, illustrated in Figure 1. In this lattice, total deterministic relations are maximal elements, hence the only way to refine them is to be equal to them. What this proposition provides is that if we have found enough relations T_i that are refined by $[w]$, and we further find that the join (T) of all the T_i relations is a total deterministic relation, we can infer that it is the function of the loop.

4.3 Equivalence Relations

Theorem 4 (and its corollaries) requires that we find transitive reflexive supersets of the loop body. In this section we provide some guidance into how such relations can be derived from a systematic analysis of the loop body. Consider a while loop of the form `while t do B` where B is written as a set of concurrent assignment statements, and consider a statement in B that has the form `x:= x+c` where x is an integer variable and c is an integer constant. We submit that by looking at this line alone we can derive a superset of $[B]$ that is reflexive and transitive. Intuitively, a reflexive, transitive relation that is a superset of B contains states s and s' such that s' can be obtained from s by an arbitrary number of applications (re: transitivity) of the loop body B, including no applications at all (re: reflexivity). Let us, without further due, submit such a relation:

$$R = \{(s, s')|x(s) \ \mathbf{mod}\ c = x(s') \ \mathbf{mod}\ c\}.$$

This relation is clearly reflexive and transitive; it is also a subset of $[B]$, since

$$x(s') = x(s) + c \Rightarrow x(s) \ \mathbf{mod}\ c = x(s') \ \mathbf{mod}\ c.$$

We can apply corollary 2 to this relation to derive a relation T that is refined by the while loop, all the while seeing very little of the loop in fact.

If we imagine that another statement of the concurrent assignment has the form `y:= y-c` for some integer variable y and for the same constant c, we can argue that because these statements are executed an equal number of times, we always preserve the sum of x and y, whence we can also derive the following reflexive transitive relation

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

then use it to derive a relation T' that is refined by $[w]$.

We want to generalize the kind of reasoning exhibited in the introductory examples presented above. To this effect, we introduce the concept of *invariant function*, which also generalizes the concept of *strongest invariant functions*, introduced in [22].

Definition 2 *Let w be the while statement* `while t do B` *on some space S, and let F be a total function on S. We say that F is an* invariant function *for w if and only if*

$$I(t) \circ [B] \circ F = I(t) \circ F.$$

Theorem 6 *If F is an invariant function for the while loop w =* `while t do B`*, and further satisfies the condition*

$$F \circ \widehat{F} \circ I(\neg t) \circ L = L$$

then $[w]$ refines the following specification

$$T = F \circ \widehat{F} \circ I(\neg t).$$

Proof. We apply theorem 4 to specification $R = F \circ \widehat{F}$. To this effect, we consider the three conditions of this theorem, which are:

1. $F \circ \widehat{F}$ is transitive,
2. $I(t) \circ [B] \subseteq R$,
3. $R \circ I(\neg t) \circ L \supseteq L$.

The first condition stems from the property that F is deterministic, as can be seen briefly:

$(F \circ \widehat{F}) \circ (F \circ \widehat{F})$

$=$ { associativity }

$F \circ (\widehat{F} \circ F) \circ \widehat{F}$

$\subseteq$ { F is deterministic }

$F \circ (I) \circ \widehat{F}$

$=$ { trivial simplification }

$F \circ \widehat{F}$.

The second condition can be established as follows. Because F is an invariant function, we can write:

$$I(t) \circ [B] \circ F = I(t) \circ F.$$

We transform this formula as follows:

$I(t) \circ [B] \circ F = I(t) \circ F$

$\Leftrightarrow$ { because $I(t) \subseteq I$ }

$I(t) \circ [B] \circ F \subseteq F$

$\Leftrightarrow$ { monotonicity of product }

$I(t) \circ [B] \circ F \circ \widehat{F} \subseteq F \circ \widehat{F}$

$\Leftrightarrow$ { because F is total, $I \subseteq F \circ \widehat{F}$ }

$I(t) \circ [B] \circ I \subseteq F \circ \widehat{F}$

$\Leftrightarrow$ { trivial simplification }

$I(t) \circ [B] \subseteq F \circ \widehat{F}.$

As for the third condition, it stems readily from the hypothesis of this theorem.

Because all three conditions of theorem 4 hold, we infer its conclusion, which is that $[w]$ refines the following specification:

$$T = (F \circ \widehat{F} \cup I) \circ I(\neg t).$$

Because F is total, its nucleus ($F \circ \widehat{F}$) is reflexive, whence we infer that

$$(F \circ \widehat{F} \cup I) = F \circ \widehat{F}.$$

Substituting in the formula of T above, we find the result of this theorem. **qed**

The importance of this theorem is that it allows us to map any invariant function that we find onto a specification that the while loop refines. Hence a loop extraction algorithm may proceed by scanning the loop body of a while loop, identifying invariant functions and mapping them to lower bounds (in the refinement ordering) of the loop function. Figure 2 shows, for illustrative purposes, the level sets of two invariant functions, and Figure 3 shows the level set of their join. Each individual invariant function partitions the domain into a set of equivalence classes; the join of the invariant functions produces the level sets of the loop function. Theorems 5 and 7 allow us to determine the image of each partition of the level set, thereby helping us to complete the definition of the loop function.

In [22] Mili et al propose a number of (strongest) invariant functions that are derived systematically from specific syntactic structures in the loop body. We broaden this list by presenting a set of invariant functions. Rather than present

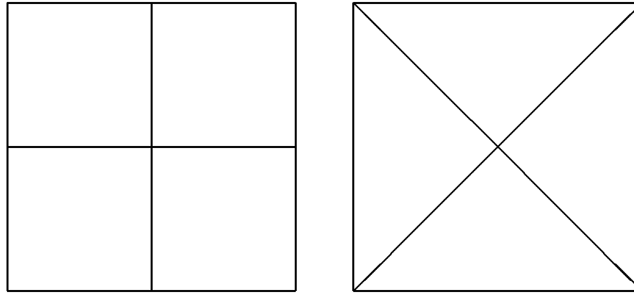

Figure 2: Invariant Functions

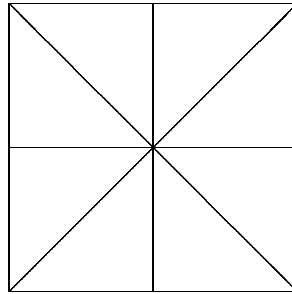

Figure 3: Strongest Invariant Function

one proposition for each invariant function, we place them in a structured fashion, using a table which highlights: the relevant syntactic structure, its corresponding invariant function, then specification T that the while statement is known to refine, by virtue of theorem 6.

Remark: On Different Forms of Loop Invariants. In [2], Basu and Misra propose that to prove the following Hoare statement valid

$$\{s = s_0\}\ \ \textbf{while}\ t\ \textbf{do}\ B\ \{s = W(s_0)\}$$

where W is the function of the loop, it is sufficient to use a loop invariant of the form

$$s \in D \wedge W(s) = W(s_0)$$

for some set D. In [20], Mili et al. consider an initialized loop of the form

$$init;\ \ \textbf{while}\ t\ \textbf{do}\ B$$

and show that set D is the invariant of Basu and Misra has the form

$$D = dom(W \cap F),$$

where F is the function of the initialized while statement. Mili et al further find that $init$ satisfies the specification

$$F\widehat{F} \cap L \circ (\widehat{W \cap F})$$

and the initial state of the loop (obtained by application of $init$) is in D. By virtue of its interpretation, D represents the set of states on which F and W compute the same value; whence we interpret Basu and Misra's loop invariant in the following terms:

$$s \in dom(W \cap F) \wedge F(s) = F(s_0),$$

or

$$(s, s_0) \in F\widehat{F} \cap (W \cap F)L.$$

Now, if we look at the loop invariants that we derive from invariant functions, they are structured as the nucleus of invariant functions. The stronger the invariant function, the stronger the loop invariant. According to [22], the loop function is a strongest invariant function, from which we derive the following strongest invariant:

$$(s, s') \in W\widehat{W}.$$

A summary comparison of the strongest loop invariant derived from Basu and Misra's formula and the strongest loop invariant derived from the loop function, yields the following observations:

- A Trivial difference: The former is a unary predicate (since s_0 is a constant) whereas the latter is a binary predicate (in s and s').
- A more meaningful difference: The former uses the nucleus of F whereas the latter uses the nucleus of W.

The second difference is all the more meaningful that F is distinct from W. We illustrate this distinction by a simple example: We consider space S defined by a variable x of type **real**, a variable a of type **array [indextype] of real** and a variable i of type **indextype**. We consider the following loop:

```
while i<>N do
   begin
   x:= x+a[i];
   i:=i+1
   end.
```

Then we find: We consider function F on S defined by:

$$F\begin{pmatrix} x \\ a \\ i \end{pmatrix} = \begin{pmatrix} \sum a[k] \\ a \\ N+1 \end{pmatrix}.$$

A theorem of Mili et al [20] provides that in order for F to be computed by a program of the form

```
init;
while i<>N do
   begin
   x:= x+a[i];
   i:=i+1
   end.
```

for some segment of code `init`, F and W must satisfy the following equation:

$$FL = (F\widehat{F} \cap L(\widehat{W \cap F}))L.$$

We admit without proof that this condition is verified, and we produce (respectively) the loop invariant derived from Basu and Misra's formula and that derived by a strongest invariant function:

$$(s, s_0) \in F\widehat{F} \cap (W \cap F)L \Leftrightarrow x = \sum_{k=1}^{i-1} a[k] \wedge a = a_0.$$

$$(s, s') \in W\widehat{W} \Leftrightarrow x + \sum_{k=i}^{N} a[k] = x' + \sum_{k=i'}^{N} a'[k] \wedge a' = a.$$

Note how profoundly distinct these two loop invariants are; the loop invariants that are useful for our purposes are of the latter kind. The distinction between these two types of invariants can be summarized (simplistically) by the observation that the former is model led by the nucleus of the initialized loop's function whereas the latter is model led by the nucleus of the (non-initialized) loop function. Another distinction is that the former establishes an asymmetric relation between the initial state (s_0) and the current state (s), whereas the latter establishes a symmetric relation between the two states s and s' (this relation treats s and s' symmetrically, and merely says that they are an arbitrary number of iterations apart).

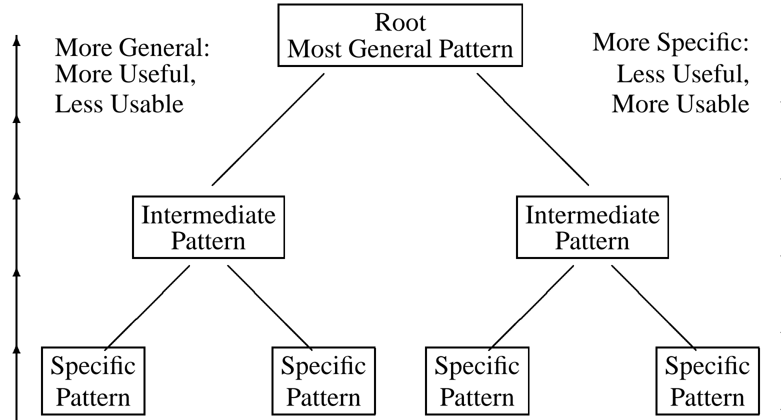

Figure 4: Pattern Tree

4.4 An Infrastructure of Syntactic Patterns

Even though we present the syntactic patterns in a monolithic tabular form, the set of patterns is best understood as a hierarchical tree structure. This structure is built on the ordering relation of generality: higher nodes in the tree represent more general patterns. More general patterns are more broadly applicable, but they are associated with lower bounds of $[w]$ that are less readily usable (require more subsequent simplification). By contrast, more specific patterns have narrower scope of application, but they are associated with lower bounds of $[w]$ that are more readily usable (require little or no simplification). This hierarchical structure is reflected in the table below by the numbering of the rules: whenever some rule A has an identifier that is a prefix of some rule B, then B is a specialization of A. The top of this hierarchy (root of the tree) is a generic pattern that views the whole loop body as a monolith; predictably, the corresponding lower bound is rather intractable, since it involves computing the transitive closure of the loop body. See Figure 4. This tree structure is used by matching a given syntactic structure of the loop with the lowest (most specific) pattern of the tree (the lowest match gives the most readily applicable lower bound). If we do not find a match with the lowest nodes of the tree, then we try matching higher and higher nodes. If we reach the node of the tree before a match is found, that may be an indication that we should add a new branch to the tree.

Id	State Space	Syntactic Pattern	Invariant Function	Lower Bound
E	x: xtype	x:=f(x)		$T = f^*\widehat{f^*} \circ I(\neg t)$
E.0	x: real; const c: real; $c \neq 0$	x:= x+c	$Fr(\frac{x}{c})$	$T = \{(s,s') \vert Fr(\frac{x}{c}) = Fr(\frac{x'}{c}) \wedge \neg t(s')\}$
E.0.0	x: int; const c: int; $c \neq 0$	x:= x+c	$x \textbf{ mod } c$	$T = \{(s,s') \vert x \textbf{ mod } c = x' \textbf{ mod } c \wedge \neg t(s')\}$
E.1	x: real; const p: int $x > 1 \wedge p > 1$	$x := x^p$	$Fr(\frac{\log(\log(s))}{\log(p)}$	$\{(s,s') \vert Fr(\frac{\log(\log(x))}{\log(p)}$ $= Fr(\frac{\log(\log(x'))}{\log(p)} \wedge \neg t(s')\}$
E.2	x: real; const c: int const m: real $m > 1 \wedge c > 0$	x:= m*x+c	$Fr(\frac{\log(x+\frac{c}{m-1})}{\log(m)}$	$\{(s,s') \vert Fr(\frac{\log(x+\frac{c}{m-1})}{\log(m)} = Fr(\frac{\log(x'+\frac{c}{m-1})}{\log(m)}$ $\wedge \neg t(s')\}$
E.3	x: real; const m: real; $m > 1$ const p: int $p > 1$;	$x := m * x^p$	$Fr(\frac{\log(\log(x)+\frac{\log(m)}{p-1})}{\log(p)})$	$\{(s,s') \vert Fr(\frac{\log(\log(x)+\frac{\log(m)}{p-1})}{\log(p)}) =$ $Fr(\frac{\log(\log(x')+\frac{\log(m)}{p-1})}{\log(p)}) \wedge \neg t(s')\}$
E.4	$x_i : real$; $1 \leq i \leq k$; $const\ m_i : real$ $1 \leq i \leq k$; $m_1 \geq 1$	$x_1 := m_1 * x_1$ $x_2 := m_2 * x_2$ $x_3 := m_3 * x_3$ ... ... $x_k := m_k * x_k$	$Fr(\frac{\log(x_1}{\log(m_1)})$, $\frac{x_2^{\log(m_1)}}{x_1^{\log(m_2)}}$, $\frac{x_3^{\log(m_1)}}{x_1^{\log(m_3)}}$, $\frac{x_k^{\log(m_1)}}{x_1^{\log(m_k)}}$,	$\{(s,s') \vert Fr(\frac{\log(x_1}{\log(m_1)}) = Fr(\frac{\log(x'_1}{\log(m_1)}) \wedge \neg t(s')\}$ $\{(s,s') \vert \frac{x_2^{\log(m_1)}}{x_1^{\log(m_2)}} = \frac{x_2'^{\log(m_1)}}{x_1'^{\log(m_2)}} \wedge \neg t(s')\}$ $\{(s,s') \vert \frac{x_3^{\log(m_1)}}{x_1^{\log(m_3)}} = \frac{x_3'^{\log(m_1)}}{x_1'^{\log(m_3)}} \wedge \neg t(s')\}$ $\{(s,s') \vert \frac{x_k^{\log(m_1)}}{x_1^{\log(m_k)}} = \frac{x_k'^{\log(m_1)}}{x_1'^{\log(m_k)}} \wedge \neg t(s')\}$

Id	State Space	Syntactic Pattern	Invariant Function	Lower Bound
E.4.0	const m: real; x: real; $m > 1$	x:= m*x	$Fr(\frac{\log(s)}{\log(m)})$	$T = \{(s,s') \| Fr(\frac{\log(x)}{\log(m)}) = Fr(\frac{\log(x')}{\log(m)}) \wedge \neg t(s')\}$
E.5	x, y: int canst a, b: int	x:=x+a, y:=y+b	$x \textbf{ mod } \|a\|$ $any - bx$	$\{(s,s') \| x \textbf{ mod } \|a\| = x' \textbf{ mod } \|a\| \wedge$ $ay - bx = ay' - bx' \wedge \neg t(s')\}$
E.6	x, y: int const a, b: int	x:=x+a y:= y+b*x	$x \textbf{ mod } \|a\|$ $y - b \times \frac{x(x-a)}{2\times a}$	$\{(s,s') \| x \textbf{ mod } \|a\| = x' \textbf{ mod } \|a\|$ $\wedge y - b \times \frac{x(x-a)}{2\times a} = y' - b \times \frac{x'(x'-a)}{2\times a}$ $\wedge \neg t(s')\}$
E.7	i: int x: xtype const c: int $c > 0$	i:=i-c; x:=f(x);	x mod c $f^{i \div c}(x)$	$T = \{(s,s') \| x \textbf{ mod } c = x' \textbf{ mod } c \wedge$ $f^{i \div c}(x) = f^{i' \div c}(x') \wedge \neg t(s')\}$
E.7.0	i: int x: xtype	i:=i-1, x:=f(x)	0 $f^i(x)$	$\{(s,s') \| f^i(x) = f^{i'}(x') \wedge \neg t(s')\}$
E.7.0.0	i: int; x: xtype const a: xtype [1..N]; const N: int	i:=i-1 x:=x+a[i],	0 $x + \Sigma_{k=1}^{i} a[k]$	$\{(s,s') \| x + \Sigma_{k=1}^{i} a[k]$ $= x' + \Sigma_{k=1}^{i'} a'[k]$ $\wedge \neg t(s')\}$
E.8	i: int x: xtype const c: int $c > 0$	i:=i+c; x:=f(x);	x mod c $f^{MaxInt-i \div c}(x)$	$T = \{(s,s') \| x \textbf{ mod } c = x' \textbf{ mod } c \wedge$ $f^{MaxInt-i \div c}(x) = f^{MaxInt-i' \div c}(x') \wedge \neg t(s')\}$
E.8.0	i: int x: type	i:=i+1, x:=f(x)	0 $f^{MaxInt-i}(x)$	$\{(s,s') \| f^i(x') = f^{i'}(x) \wedge \neg t(s')\}$

Id	State Space	Syntactic Pattern	Invariant Function	Lower Bound
E.9	x1: x1type	$x_1 = E_x^1$,	0	
	x2: x2type ...	$x_2 = E_x^2$,	0	
	x_k: $x_k type$	$x_k = E_k^1$,	0	
	x_{k+1}: $x_{k+1} type$ ...		x_{k+1}	$T = \{(s,s') \mid x_{k+1} = x'_{k+1}$
	x_n: $x_n type$		x_n	$\wedge x_n = x'_n\}$
E.10	i: int	i:=i-1	0	$T = \{(s,s') \mid f \oplus \bigoplus_{k=1}^{i} k = f' \oplus \bigoplus_{k=1}^{i'} k$
	a: arraytype	$a := a \oplus i$	$f \oplus \bigoplus_{k=1}^{i} k$	$\wedge \neg t(s')\}$

4.5 Rectangular Relations

In the previous subsection, we have explored transitive relations which are derived as nuclei ($F \circ \widehat{F}$) of invariant functions (F). In this section, we consider transitive relations which are rectangular, rather than equivalence relations. We have the following theorem, which is a special corollary of theorem 4.

Theorem 7 *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 the function of this while statement ($[w]$) refines specification T, where T is defined as:*

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

Proof. We use theorem 4, in which we submit that $L \circ [B]$ satisfies the required conditions on R. To this effect, we consider in turn all the conditions of theorem 4.

- *R is total.* Because t is not the trivial condition **false** , the while statement invokes its loop body for at least some initial state. Because the while statement w terminates for all initial states, $[B]$ is necessarily non-empty. By Tarski's identity, we can infer

  $$L \circ [B] \circ L = L.$$

  Substituting $L \circ [B]$ by R, we find

  $$RL = L,$$

which provides that R is total.

- R *is transitive*. We compute RR and prove that it is a subset of R.

RR

$=$ { substitution }

$(L \circ [B]) \circ (L \circ [B])$

$=$ { associativity }

$(L \circ [B] \circ L) \circ [B]$

$=$ { $[B]$ is not empty, Tarski's identity }

$L \circ [B]$

$=$ { substitution }

R.

- $R \circ I(\neg t) \circ L = L$. By substituting R, we find that this condition is equivalent to

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

By Tarski's identity, this condition holds if and only if

$$[B] \circ I(\neg t)$$

is non-empty. We find that the conditions of this theorem preclude that this expression be empty. Because t is not the trivial condition **false** , the while statement invokes its loop body for at least some initial state. If the expression

$$[B] \circ I(\neg t)$$

were empty, it means that testing condition t after execution of loop body B would always yield **true** . Whence execution of the while statement on any state that causes the loop body to be invoked would lead to an infinite loop (since whenever we execute B then test t, we find it to be true). This contradicts the hypothesis that the while statement terminates for all its initial states.

qed

The conditions of this theorem exclude the pathological case where a loop never invokes its loop body (if t is $\mathbf{false}$) and the pathological case where the loop never terminates (if t is guaranteed to be $\mathbf{true}$ by the prior execution of B). Excluding these two cases, this theorem provides a lower bound (in the refinement ordering) of the loop function by analyzing the function of the loop body, specifically to derive its range. The expression $L \circ [B]$ can be written simply as:

$$L \circ [B] = \{(s, s')|s' \in rng([B])\}.$$

Hence to apply this theorem, we need to compute the range of $[B]$. The proposition below, whose proof mimics trivially that of theorem 7, provides that if the range of $[B]$ is too difficult to compute, we can approximate it with upper bounds.

Proposition 5 *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** *, and given a total rectangular relation R that is a superset of $L \circ [B]$. Then the function of this while statement ($[w]$) refines specification T, where T is defined as:*

$$T = (R \cup I) \circ I(\neg t).$$

To illustrate theorem 7, we consider the following while loop, where x, y and z are integer variables:

```
w =
while z<>1 do
    [
     x:= x+2;
     y:= y-3;
     z:= x+y
    ]
```

We find (by inspection) that this loop terminates if and only if

$$z = 1 \vee x + y \geq 1.$$

Indeed, if $z <> 1$, then we iterate at least once, upon which z takes the value of $(x+y)$; subsequent iterations (if any) place $(x+y)$ into z at each iteration and test the resulting value of z to determine termination; the sum $(x+y)$ decreases by one at each iteration, and eventually becomes 1 only if the original value of $(x+y)$ is greater than or equal to 1. Hence we redefine the space of the program to be the set of values of integer variables x, y and z, such that

$$z = 1 \vee (x+y) \geq 1.$$

On this space, the loop terminates, and the loop function is total.

We apply pattern E.5 to the first two lines of the loop body, and infer that $[w] \sqsupseteq T1$, where

$$T1 = \{(s,s')|x \ \mathbf{mod}\ 2 = x' \ \mathbf{mod}\ 2 \wedge 3x + 2y = 3x' + 2y' \wedge z' \leq 1\}.$$

No other combination of lines matches the table of patterns, hence we turn to theorem 7 for further analysis. This theorem requires that we compute the range of $[B]$.

$L \circ [B]$

$=$ { Interpretation of relational product }

$\{(s,s')|\exists s'' : (s,s'') \in L \wedge (s'',s') \in [B]\}$

$=$ { simplification, since L is universal relation }

$\{(s,s')|\exists s'' : (s'',s') \in [B]\}$

$=$ { substitution of $[B]$ }

$\{(s,s')|\exists s'' : x' = x'' + 2 \wedge y' = y'' - 3 \wedge z' = x'' + y''\}$

$=$ { arithmetic substitution }

$\{(s,s')|\exists s'' : x' = x'' + 2 \wedge y' = y'' - 3 \wedge z' = x' - 2 + y' + 3\}$

$=$ { arithmetic simplification, logical simplification }

$\{(s,s')|z' = x' + y' + 1 \wedge \exists s'' : x' = x'' + 2 \wedge y' = y'' - 3\}$

$=$ { logical simplification (since x and y are integer variables) }

$\{(s,s')|z' = x' + y' + 1\}.$

We notice that the loop body is not surjective, whence we propose to apply theorem 7. This theorem provides that $[w]$ refines relation $T2$, where

$$T2 = \{(s,s')|z' = x' + y' + 1 \wedge z' \leq 1\} \cup I(z \leq 1).$$

Taking the join of these two relations, we find:

$[w]$

$\sqsupseteq$ { Lattice Property }

$T1 \sqcup T2$

$=$ { substitution, simplification }

$\{(s,s')|z \neq 1 \wedge x \,\mathbf{mod}\, 2 = x' \,\mathbf{mod}\, 2 \wedge 3x + 2y = 3x' + 2y' \wedge z' = x' + y' + 1 \wedge z' = 1\} \cup I(z = 1).$

$=$ { merging two expressions of z' }

$\{(s,s')|z \neq 1 \wedge x \,\mathbf{mod}\, 2 = x' \,\mathbf{mod}\, 2 \wedge 3x + 2y = 3x' + 2y' \wedge 0 = x' + y' \wedge z' = 1\} \cup I(z = 1).$

$=$ { highlighting the form $(x' + y')$ }

$\{(s,s')|z \neq 1 \wedge x \,\mathbf{mod}\, 2 = x' \,\mathbf{mod}\, 2 \wedge 3x + 2y = x' + 2(x' + y') \wedge 0 = x' + y' \wedge z' = 1\} \cup I(z = 1).$

$=$ { substitution }

$\{(s,s')|z \neq 1 \wedge x \,\mathbf{mod}\, 2 = x' \,\mathbf{mod}\, 2 \wedge 3x + 2y = x' \wedge 0 = x' + y' \wedge z' = 1\} \cup I(z = 1).$

$=$ { inferring an expression for y' }

$\{(s,s')|z \neq 1 \wedge x \,\mathbf{mod}\, 2 = x' \,\mathbf{mod}\, 2 \wedge 3x + 2y = x' \wedge y' = -3x - 2y \wedge z' = 1\} \cup I(z = 1).$

$=$ { highlighting the difference between x and x' }

$\{(s,s')|z \neq 1 \wedge x \,\mathbf{mod}\, 2 = x' \,\mathbf{mod}\, 2 \wedge x' = x + 2(x + y) \wedge y' = -3x - 2y \wedge z' = 1\} \cup I(z = 1).$

$=$ { because x and x' are a multiple of 2 apart, they have the same remainder by 2 }

$\{(s,s')|z \neq 1 \wedge x' = x + 2(x + y) \wedge y' = -3x - 2y \wedge z' = 1\} \cup I(z = 1).$

This is the union of two functions, whose domains are disjoint; it is a function. Furthermore, it is a total function, since the union of the domains of the terms ($\{s|z \neq 1\}$ and $\{s|z = 1\}$) is S. Whence we infer

$$[w] = \{(s, s')|z \neq 1 \wedge x' = x + 2(x + y) \wedge y' = -3x - 2y \wedge z' = 1\} \cup I(z = 1).$$

4.6 Other Forms of Transitive Supersets?

Theorem 4 provides a general framework for deriving lower bounds of the loop function, which theorems 6, 5 and 7 specialize to produce equivalence relations (in the case of theorem 6) and rectangular relations (in the case of theorems 5 and 7). This raises the question: are these theorems sufficient for our purposes? We submit two arguments to the effect that they appear to be.

First, a theorem by S.K. Basu and J. Misra [2] provides that in order for w to compute F, it has to have a loop invariant of the form

$$s \in D \wedge F(s_0) = F(s),$$

for some subset D of S. We rewrite this as

$$(s_0, s) \in \Delta \wedge (s_0, s) \in \Phi,$$

where $\Delta = S \times D$ and $\Phi = F \circ \widehat{F}$. We observe that Δ is rectangular, and Φ is built as the nucleus of an invariant function (according to [22], the function of w is an invariant function for w).

Second, we consider that one way to derive a total function from S to S is to proceed in two steps (see Figure 5):

- First, determine how the function partitions its domain (all of S) into level sets. Theorem 6 allows us to do so in a stepwise manner, by identifying arbitrarily large equivalence relations that are preserved by the loop body.

- Second, determine what image the function assigns to each level set. Theorems 5 and 7 enable us to do so; the former is useful whenever the negation of the loop condition is not sufficient to characterize the final states, and the latter is useful whenever the loop body is not surjective.

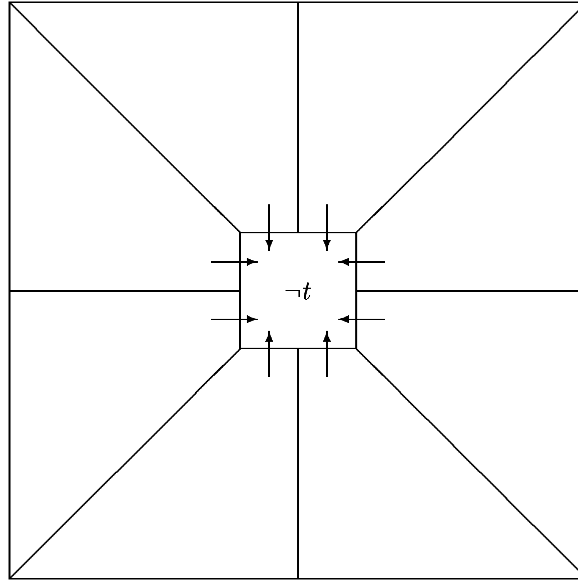

Figure 5: Function Derivation

5 Outline of a Loop Extraction Algorithm

Based on the foregoing discussions, we are currently in the preliminary stages of developing a prototype for the extraction of loop functions. As we envision it, the prototype proceeds through the following phases:

- Reduce the space S so that $dom([w]) = S$. We know from proposition 2 that this does not cause a loss of generality. Nevertheless, this is a non-trivial step, whose automation we are still exploring. Until this step is automated, we only submit loops whose space is restricted to their domain.

- The prototype has at its disposal a database of *Semantics Recognizers* (abbr: recognizers), which are characterized by the following features: a state space (represented by set of variable declarations); a syntactic pattern (represented by a set of concurrent assignments); and a lower bound (represented by a relation). The prototype scans the loop body of the object loop (which is represented by concurrent assignments) looking for matches between actual source code and syntactic patterns. Whenever a match occurs between the state space and syntactic pattern of a recognizer and the source code of the loop of interest, the lower bound of the recognizer is instantiated with the appropriate substitutions (replacing symbolic variables of the recognizer with actual loop variables) and an actual lower bound is generated.

- As a default option, we automatically generate the lower bounds provided by theorems 5 and 7. This may be redundant is some cases, but it is simpler than having to recognize cases where it is necessary.

- The set of lower bounds is passed on to *Mathematica* (©Wolfram Research), which infers from it explicit values of the primed variables (or returns parts of it unresolved if the lower bounds that we have provided do not define a deterministic relation). Specifically, the equations produced from the lower bounds have the following form:

$$\begin{pmatrix} f_1(x, y, z, ...) = f_1(x', y', z', ...) \\ f_2(x, y, z, ...) = f_2(x', y', z', ...) \\ f_3(x, y, z, ...) = f_3(x', y', z', ...) \\ ... \\ p(x', y', z') \end{pmatrix},$$

  where the symmetric equations stem from the recognizer lower bounds and the unary equations (in the primed variables) stem from lower bounds provided by theorems 5 and 7. Using functions `Simplify` and `Reduce`, Mathematica maps these into a functional representation, of the form:

$$\begin{pmatrix} x' = E_x(x, y, z, ...) \\ y' = E_y(x, y, z, ...) \\ z' = E_z(x, y, z, ...) \\ ... \end{pmatrix},$$

  which is the form that a user may want to read. 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 data types.

A detailed specification (in Z) of the data types and algorithms of this loop extraction prototype is given in [21].

6 Sample Examples

In this section, we briefly illustrate the proposed approach on a few simple examples, using the recognizers that are shown in section 4. The proposed method has no intrinsic limitation; it is as powerful as the recognizers that are

available to it. We are currently exploring more advanced recognizers, dealing with more complex statements, and more complex domain abstractions.

6.1 Arithmetic Manipulations

We consider the following loop, where all the variables are of type integer. The loop body is a set of concurrent assignments, which are understood to apply concurrently.

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

We consider the first two assignments and apply pattern E.5, with $a = -1$ and $b = +2$. We find the following invariant function, that we call $F1$:

$$F1\begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} = \begin{pmatrix} 0 \\ 2x+y \\ 0 \\ 0 \end{pmatrix}.$$

From this function, we derive specification $T1$ defined as

$$T1 = \{(s, s')|2x + y = 2x' + y' \wedge x' \leq 0\}.$$

We apply pattern E.6 to the first and third statement, with $a = -1$ and $n = 1$, and find the following invariant function, which we call $F2$:

$$F2\begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ z + \frac{x(x+1)}{2} \\ 0 \end{pmatrix}.$$

From this function, we derive specification $T2$ defined as

$$T2 = \{(s,s')|z + \frac{x(x+1)}{2} = z' + \frac{x'(x'+1)}{2} \wedge x' \leq 0\}.$$

We apply pattern E.6 to lines 2 and 4, with $a = +2$ and $b = 2$ and find the following function, which we call $F3$:

$$F3 \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} = \begin{pmatrix} 0 \\ y \ \mathbf{mod}\ 2 \\ 0 \\ w - \frac{y(y-2)}{2} \end{pmatrix}.$$

From this function, we derive specification $T3$ defined as

$$T3 = \{(s,s')|y \ \mathbf{mod}\ 2 = y' \ \mathbf{mod}\ 2 \wedge w - fracy(y-2)2 = w' - fracy'(y'-2)2 \wedge x' \leq 0\}.$$

According to theorem 4, $[w]$ refines $T1$, $T2$ and $T3$, which we write

$$[w] \sqsupseteq T1 \wedge [w] \sqsupseteq T2 \wedge [w] \sqsupseteq T3.$$

By lattice theory, we infer

$$[w] \sqsupseteq T1 \sqcup T2 \sqcup T3.$$

By Proposition 3, this join is defined, equals the intersection of the terms, and is total. We find

$$[w] \sqsupseteq T1 \cap T2 \cap T3.$$

Unfortunately, the right hand term in this equation is not a deterministic relation. The reasons is that all we know about the final value of x is that it is less than or equal to 0[1]. Theorem 5 comes to the rescue; this theorem provides that not only does the final state not satisfy t, but it is the first state (in the iteration sequence) not to satisfy t; this should help us establish that the final value of x is 0 (so far we only know it is less than or equal to 0). We satisfy the conditions of theorem 5 because the condition of the loop is not trivially false. Whence we infer

$$[w] \sqsupseteq T4,$$

where

$$T4 = I(t) \circ L \circ I(t) \circ [B] \circ I(\neg t) \cup I(\neg t).$$

[1]Had the condition of the loop been $x \neq 0$ rather than $x > 0$, we would have inferred that $x' = 0$.

Without getting into much detail, we find that $T4$ can be written (by definition) as

$$T4 = \{(s,s')|x > 0 \wedge x' + 1 > 0 \wedge x' \leq 0\} \cup \{(s,s')|x \leq 0 \wedge s' = s\}.$$

By simplifying the first term of this expression, we find

$$T4 = \{(s,s')|x > 0 \wedge x' = 0\} \cup \{(s,s')|x \leq 0 \wedge s' = s\}.$$

Now, we let T be the join of $T1, T2, T3$, and $T4$, and we find that

$$T = \{(s,s')|x > 0 \wedge x' = 0 \wedge y' = 2x + y \wedge z' = z + \frac{x(x+1)}{2} \wedge w' = w + 2x(y + x - 1)\}.$$

We notice that T is a total function, whence, according to Proposition 4, the statement

$$[w] \sqsupseteq T$$

allows us to infer

$$[w] = T.$$

6.2 Array Manipulations

We consider the following loop program

```
w =
while i <> 0 do
    {
    x:= x+a[i],
    i:=i-1
    }
```

on a space S defined by

```
    x:  itemtype;
    a:  array [1..N] of itemtype;
    i:  0..N;
```

By inspection, we can determine that w terminates for all s in S. Hence we admit that $[w]$ is total on S. We apply pattern E.9 to this loop, yielding the following invariant function

$$F1\begin{pmatrix} i \\ x \\ a \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ a \end{pmatrix}.$$

From this invariant function, we infer the following T specification

$$T1 = \{(s, s')|a = a' \wedge i' = 0\}.$$

We infer that we meet the conditions of pattern E.7.0.0, whence we find that $[w]$ refines the following specification:

$$T2 = \{(s, s')|x + \sum_{k=1}^{i} a[k] = x' + \sum_{k=1}^{i'} a'[k] \wedge i' = 0\}.$$

Taking the join of these specifications, we find:

$$T = \{(s, s')|a = a' \wedge x + \sum_{k=1}^{i} a[k] = x' + \sum_{k=1}^{i'} a'[k] \wedge i' = 0\}.$$

We simplify this into (given that the sum starting at 1 and ending at 0 is trivially 0):

$$T = \{(s, s')|a = a' \wedge x + \sum_{k=1}^{i} a[k] = x' \wedge i' = 0\}.$$

This relation is deterministic as it specifies a final value for all three variables of the state space; in addition, it is total (for all s, it is possible to compute an image s'). We infer that T is maximal in the refinement ordering, whence, since $[w]$ refines T, we infer that $[w]$ equals T.

In the second example of array manipulations, we illustrate pattern E.10, which captures cell by cell operations on arrays. To this effect, we consider the following while loop

```
while (i<>0) do
    {
    a[i]:=0,
    i:= i-1
    }
```

on space S defined by

```
i: 0..N;

a: array [1..N] of itemtype;

const N: int;
```

We interpret the assignment `a[i]:=0` as an operation on the whole array a, whereby the right hand side is an expression involving (generally) a and i, and the right hand side is array a. Whence we match it against the pattern

$$a := a \oplus i.$$

Other assignments that match the same pattern include:

```
a[i] := a[i]+1;

a[i] := a[i]+i;

a[i] := i;

a[i] := 2*a[i];

... ... ...
```

Application of this pattern yields that $[w]$ refines the following specification:

$$T1 = \{(s, s')|a \oplus \bigoplus_{k=1}^{i} k = a' \oplus \bigoplus_{k=1}^{i'} k \wedge i' = 0\}.$$

The axiomatization of operation $\bigoplus$ provides that the sequential application of this operation for indices k ranging from 1 to ... zero does not alter the argument. Whence this can be simplified to:

$$T1 = \{(s, s')|a' = a \oplus \bigoplus_{k=1}^{i} k \wedge i' = 0\}.$$

This is a total deterministic relation, hence it is maximal by refinement. Whence we infer:

$$[w] = \{(s, s')|a' = a \oplus \bigoplus_{k=1}^{i} k \wedge i' = 0\}.$$

We interpret this formula as: array a' is obtained from array a by assigning 0 to all the cell between indices 1 and i (while preserving the other cells); and index i' is assigned value 0.

6.3 String Manipulations

Let f be a variable of type *string*, c be a variable of type *char* (or string, in fact) and i be a variable of type integer. We consider the following while loop on space S defined by these variables.

```
while (i<>0) do
    {
    i:= i-1,
    f:= f.c
    }
```

where the dot represents string concatenation. We interpret c as a numeral representation, in base q, say (the base depends on the width of c), and we interpret the concatenation `f.c` numerically as

$$q * f + c$$

where f and c now represent (by abuse of notation) the numeric values of strings f and c. Whence the loop body can now be reinterpreted as

```
i:= i-1,
f:= q*f+c
```

where q is a constant. Using a change of variable (that which allowed us to derive rule E.2) we can bring this loop body to match the pattern E.5. The resulting lower bound is then

$$T1 = \{(s, s')|i' = 0 \wedge f' = f \times q^i + c \times \frac{q^i - 1}{q - 1}\}.$$

It is a well known identity that

$$\frac{q^i - 1}{q - 1} = q^{i-1} + q^{i-2} + ... + q^2 + q + 1.$$

In base q, this can be written as the following sequence of length i:

$$111....111.$$

Multiplying by c, where c is less than q, we find the following sequence of length i:

$$ccc....ccc.$$

Adding this to $f \times q^i$ we find (in base q):

$$fccc....ccc,$$

which is indeed the string assigned to f'.

7 Conclusion

7.1 Summary

In this paper we have analyzed while loops with the intent of supporting the stepwise derivation of loop functions. The function of a loop of the form

```
while t do B
```

has an explicit formula, as per the following equation:

$$[w] = (I(t) \circ [B])^* \circ I(\neg t).$$

To extract the function of this loop means, in effect, to derive a closed form expression for the reflexive transitive closure term,

$$(I(t) \circ [B])^*.$$

The core idea of this paper is to derive this transitive closure in a stepwise manner, by deriving arbitrary transitive supersets of it. We have explored two special forms of transitive supersets: Equivalence relations derived as nuclei of invariant functions; rectangular relations characterizing unary effects of loop bodies. We have also briefly discussed the outlines of an algorithm that implements this approach to derive the function of a loop automatically.

7.2 Assessment

The main contributions of this paper are listed as follows:

- A lattice based approach that provides for deriving the loop function by a succession of lower bounds; this approach's main advantage is its ability to deal with potentially large and complex loops by tackling a few lines of code at a time. Another advantage is that even when the proposed solution cannot derive the full function of the loop, it extracts functional information about it, that is commensurate with the amount of code patterns it could recognize.

- A set of theorems and corollaries, all of which are original (to the best of our knowledge), that provide lower bounds of the loop function, under various conditions.

- The outline of an algorithm for deriving loop functions automatically, using a database of semantic recognizers. The design of this database determines to a large extent the effectiveness (scope of application) of the proposed algorithm. Also, the deployment of *Mathematica* (©Wolfram Research) to solve the symbolic equations produced by the recognizers makes us hopeful that we can produce a working prototype al relatively low cost, given that the matching step is fairly straightforward.

The problem of deriving the function of a loop is admittedly very difficult in general; also, the stakes attached to solving this problem are very high. We estimate that the approach that we propose is fairly broad in its outline, even though the implementation we are currently envisioning is limited.

7.3 Prospects of Further research

The ideas presented in this paper are preliminary; we envision to expand them in the following directions.

- *Theoretical Extensions*. While the results we have presented in section 3 are fairly general, the solutions we proposed in section 4.3 are rather restricted. Their main limitation is that they apply exclusively to concurrent assignment statements that are not conditional. The first theoretical extension that we envision is to derive rules for finding lower bounds of the loop function even when the concurrent assignments of the loop body are conditional.

  What has made it possible for us to derive reflexive transitive supersets of the loop body is that in the absence of conditionals, concurrent assignments represent supersets of the loop body. With the introduction of <u>conditional</u> concurrent assignments, individuals assignments are partitions of (i.e. subsets of) superssets of the loop body. One venue we envision to explore in this case is to analyze formulas for deriving the transitive closure of a union.

- *Practical Extensions*. The main practical extension is, obviously, the derivation of the prototype for extracting loop functions. Orthogonally with the development of the prototype per se, we envision to evolve the database of recognizers, so as to enhance its extraction effectiveness. To this effect, we are depending not only on the theoretical results above, but also on the active research on loop invariants that is discussed in [4, 5, 7, 8, 16–18, 27, 28]. We envision to use the insights gained in this type of research to derive recognizers for more

complex while loops. Another trivial extension is to extend the front end of the prototype by a function that maps programming language source code (such as C, C++, Java) into concurrent assignment statements; such a features would allow users to run the prototype on their source code directly. Current compiler generation technology makes this a straightforward task.

This is clearly a tall order; as we discussed in the introduction, the stakes of this research make it very worthwhile.

References

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

[2] S.K. Basu and J.D. Misra. Proving loop programs. *IEEE Transactions on Software Engineering*, 1(1):76–86, 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. Rodriguez Carbonnell and Deepak 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] Rosann W Collins, Gwendolyn H. Walton, Alan R Hevner, and Richard 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.

[7] 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.

[8] 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.

[9] Ewen Denney and Bernd 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.

[10] E.W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.

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

[12] Michael D. Ernst, Jeff H Perkins, Phillips J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 2006.

[13] Thomas Fahringer and Bernhard Scholz. *Advanced Symbolic Analysis for Compilers*. Springer Verlag, Berlin, Germany, 2003.

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

[15] Alan R Hevner, Richard C Linger, Rosann W Collins, Mark G Pleszkoch, Stacy J. Prowell, and Gwendolyn 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.

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

[18] T. Jebelean L. Kovacs. An algorithm for automated generation of invariants for loops with conditionals. In D. Petcu 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.

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

[20] A. Mili, J. Desharnais, and F. Mili. Relational heuristics for the design of deterministic programs. *Acta Informatica*, 24(3):239–276, 1987.

[21] Ali Mili, Tim Daly, Mark Pleszkoch, and Stacy Prowell. A semantic recognizer infrastructure for computing loop behavior. In *Proceedings, 40th Annual Hawaii International Conference on Systems Sciences*, Big Island, HI, January 2007.

[22] Ali Mili, Jules Desharnais, and Jean Raymon Gagne. Strongest invariant functions: Their use in the systematic analysis of while statements. *Acta Informatica*, April 1985.

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

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

[25] J.H. Morris and B. Wegbreit. Program verification by subgoal induction. In R.T. Yeh, editor, *Current Trends in Programming Methodology*, volume II, chapter 8. Prentice Hall, Englewood Cliffs, NJ, 1977.

[26] Linda Northrop, Peter Feiler, Richard P. Gabriel, John Goodenough, Rick Linger, Tom Longstaff, Rick Kazman, Mark Klein, Douglas Schmidt, Kevin Sullivan, and Kurt Wallnau. *Ultra large Scale Systems: The Software Challenge of the Future*. Software Engineering Institute, July 2006.

[27] 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.

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

[29] A. Tarski. On the calculus of relations. *The Journal of Symbolic Logic*, 6(3), September 1941.

[30] A. Tarski. A lattice-theoretic fixpoint theorem and its applications. *Pacific Journal of Mathematics*, (5):285–309, 1955.