Toward the Automated Derivation of Loop Functions

Ali Mili¹, Shir Aharon¹, Mark Pleszkoch², Richard C. Linger²
¹: College of Computer Science, New Jersey Institute of Technology, Newark NJ 07102-1982
mili@cis.njit.edu, ShirAharon09@comcast.net
²: Software Engineering Institute, Carnegie Mellon University, Pittsburgh PA 15213-3890
{rlinger,mpleszko}@sei.cmu.edu

September 6, 2007

Abstract

Modern software applications are growing increasingly large and complex, at the same time as they are taking on increasing critical roles. To deal with this twofold pressure, we need automated tools that help us analyze the function of programs or program fragments to an arbitrary level of precision. This paper discusses an approach to computing the function of a while loop, and illustrates it by means of a simple prototype.

Keywords

Functional extraction, loop function, sub-goal induction theorem, Mills’ theorem, loop invariants, refinement calculus, relational mathematics, program comprehension.

1 Introduction: Motivation and Premises

Modern software applications are taking increasingly critical roles, placing higher and higher stakes in their validation and certification. Simultaneously, recent software paradigms, such as reuse, product line engineering, and outsourcing, involve using or reusing software whose development we have not monitored; this tends to shift the burden of verification and certification to post-development analysis (rather than monitoring development processes, or enforcing development standards). Finally, the size and complexity of current software applications makes it impossible to verify or certify such applications to the required quality standards without automated support. These premises, together, make a case for the development of tools that help us analyze source code to an arbitrary degree of thoroughness and precision. Such tools would have an impact on software verification, software inspection, software maintenance, and software reuse.

In this paper, we present some theoretical results that allow us to derive the function of a while loop in an imperative programming language (C, C++, Java), and we briefly illustrate these results by deploying a prototype that implements the theoretical results. Specifically, we consider a loop of the form

while t do {...}

on some variables x, y, z, etc. and we are interested in deriving the function that this loop defines between the initial values of its variables, and their final values.

1.1 Introductory Examples

In order to help the reader gain a clear idea of our goal, and to reflect the current capability of our proposed algorithm, we present below a set of four simple loops, along with the functions that our algorithm computes for them. We use C++ syntax to represent the original loop, and we use mathematical (relational) notation to represent the loop function. Though the examples are fairly simple, the algorithm is not limited, in the sense that we can evolve it to deal with a wide range of data types and operations, by merely adding new knowledge, as we discuss in the sequel (section 4).
1.1.1 A Numeric Example

The first example involves numeric computations. We introduce a number of constants, making this a family of programs (according to the values of constants), rather than a single program (because for some values of the constants the shape of the loop changes). We consider the following C++ Program:

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

int x, t, i, v, w, y, z; // program variables
const int a = ..; // program constants
const int b = ..;
const int c = ..;
const int d = ..;
const int e = ..;

int main ()
{
    while (i != 0)
    {
        v = v + a*t;
        z = z + c*x;
        w = w + e*y;
        x = x+a;
        y = y+b;
        t = t*d;
        i = i-1;
    }
}
```

We are interested in deriving the function that this loop defines between its initial states (values of $x, y, z, t, v, w$ and $i$ prior to the execution of the loop) and its final states (values of these variables when execution terminates).

The function of this loop is actually more complicated than may appear, because of the variety of configurations of the constants in this program ($a, b, c, d, e$). For example, if constant $a$ is zero, then variables $v$ and $x$ are preserved, and the expression of $z$ becomes a multiplication rather than the sum of an arithmetic series. Likewise, if constant $d$ is equal to 1 then variable $t$ is preserved and the expression of $v$ becomes a multiplication rather than the sum of a geometric series. Our algorithm produces the function of this loop as a union of several terms, one for each possible configuration of the values of the program constants. For the sake of simplicity, we present below a set of three representative terms. The first one reflects the case where all the constants have non-trivial values and the loop iterates at least once; the second one reflects the case where the loop does not iterate at all; and the third term reflects the case where constant $e$ is zero.

\[
\begin{bmatrix}
    x & x' \\
    y & y' \\
    z & z' \\
    v & v' \\
    w & w' \\
    t & t' \\
    i & i'
\end{bmatrix}
\]

\[
\left\{ \begin{array}{l}
\frac{\text{d} \neq 1 \land abcde \neq 0 \land i' = 0 \land t' = \frac{\text{d}t + \text{e}t - \text{a}t - \text{v}}{\text{(d-1)}} \land \frac{\text{u}'}{2} = \frac{\text{be}t - \text{bi} + 2\text{c}t}{2} \land \frac{x'}{2} = \text{x} + \text{ai} \land \frac{y'}{2} = \text{y} + \text{bi} \land \frac{z'}{2} = \frac{\text{ac}t - \text{ae}t + \text{cex}t + \text{z}}{2} } \end{array} \right\}
\]
In order to gain some confidence in the correctness of this function, we have set up a test driver that invokes the loop on various initial data and checks the output against a test oracle derived from the function. The test driver checked the loop against the oracle for about a quarter billion test cases (244,140,625); all the tests were successful. We generated these test data by varying the initial values of the program variables and program constants within prescribed ranges (avoiding value 0 for $a$, $b$, $c$, and values 0 and 1 for $d$).

1.1.2 A Function Call

In this example we want to showcase our algorithm’s ability to deal with abstract function calls, provided the user / analyst is willing to get the result in terms of the function in question. We consider the following C++ program:

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

int x, y, z, j, t, v, w;
int f (int v);

int main ()
{
    while (x>5)
    {
        x = x-5;
        y = y+b;
        z = z+b*x+5*b;
        j = j-1;
        t = t+c*y-b*c;
        v = f(v);
        w = w+v;
    }
}

int f (int v)
{
    // the actual definition of f does not matter
    // for our purposes, since the result is given
    // in terms of f.
}
Our algorithm provides the following function for this loop.

\[
\begin{pmatrix}
  x & x' \\
  y & y' \\
  z & z' \\
  v & v' \\
  w & w' \\
  t & t' \\
  j & j'
\end{pmatrix}
\]

\[
x' = x \mod 5 \land
y' = \frac{b \times 5 y - b(x \mod 5)}{5} \land
z' = z + \frac{b \times 5 y - b(x \mod 5)}{5} \land
v' = f^{x/5}(v') \land
w' = w + \sum_{k=1}^{x/5} f^k(v).
\]

To gain confidence in the validity of the proposed function, we have tested the loop in question against the oracle derived from the function definition above, using nearly 30 million test cases, and all have returned true (with a sample function \( f \) defined by \( f(v) = 2 \times v + 1 \)). The ability of our algorithm to handle abstract functions (i.e. functions that it does not need to analyze) affords us a measure of abstraction.

1.1.3 Array processing

This example showcases our algorithm’s ability to deal with arrays. We consider a loop that scans two arrays, in opposite directions.

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

const int N = ..;
int a[N]; int b[N];
int x, y, i, j;

int main ()
{
  while (i != N+1)
  {
    x = x + a[i];
    y = y + b[j];
    i = i+1;
    j = j-1;
  }
}
```

Our algorithm finds the following function for this program:

\[
\begin{pmatrix}
  x & x' \\
  y & y' \\
  i & i' \\
  j & j' \\
  a & a' \\
  b & b'
\end{pmatrix}
\]

\[
a' = a \land b' = b \land i' = i + j - (N + 1) \land
x' = x + \sum_{k=i}^{N} a[k] \land
y' = y + \sum_{k=1}^{j} b[k] - \sum_{k=1}^{i+j-(N+1)} b[k].
\]

Given that \( i \) is an index to the array, it is necessarily less than \( N + 1 \), whence \( i + j - (N + 1) < j \). Therefore, we can further simplify the expression of this function:

\[
\begin{pmatrix}
  x & x' \\
  y & y' \\
  i & i' \\
  j & j' \\
  a & a' \\
  b & b'
\end{pmatrix}
\]

\[
a' = a \land b' = b \land i' = i + j - (N + 1) \land
x' = x + \sum_{k=i}^{N} a[k] \land
y' = y + \sum_{k=i+j-N}^{j} b[k].
\]
We have tested this loop against the oracle derived from this function definition, using a quarter million test data (by varying values of the various variables, including the contents of the array); all were successful.

### 1.1.4 List Processing

This example showcases the ability of our algorithm to handle non-scalar abstract data types, provided their axiomatization is taken into account. We consider the following C++ program:

```cpp
#include <list>
#include <iostream>
using namespace std;
list <int> x;
list <int> y;
int i; int t;
const int c=5;
int main ()
{
    while (!(x.empty()))
    {
        i = i-1;
        y.push_back(x.front());
        x.pop_front();
        t = t*c;
    }
}
```

The loop function we find is:

\[
\{ (x^{i}, x') \mid x' = \{ \} \wedge y' = y \wedge i' = i - \text{length}(x) \wedge t' = t \cdot \text{length}(x) \}. 
\]

The reader can easily convince herself/himself that this is indeed the function of the loop at hand, where we use the dot (.) to represent list concatenation. The number of iterations of the loop equals the length of list \(x\), and at each iteration, an element is extracted from the front of \(x\) and added to the back of \(y\). Independently, each iteration subtracts 1 from \(i\) and multiplies \(t\) by \(c\).

If we change the control of the loop to be testing for zero on the integer variable \(i\), then the program becomes:

```cpp
#include <list>
#include <iostream>
using namespace std;
list <int> x;
list <int> y;
int i; int t;
const int c=5;
while (!(i==0))
{
    i = i-1;
    y.push_back(x.front());
    x.pop_front();
    t = t*c;
}
```
The loop function we find is:

\[
\{ (x, x') \mid y \leq \text{length}(x) \land x' = \text{Rest}^i(x) \land y' = \text{Rest}^i(y) \land x . x' = y . x' \land t' = tc^i \}.
\]

This time, the loop executes \(i\) times, assuming that \(i\) does not exceed the length of \(x\). When it completes its execution, the final value of variable \(i\) is zero, the final value of variable \(x\) is the list obtained by truncating list \(x\) \(i\) times, and the final value of variable \(y\) is obtained by appending to \(y\) all the values that have been truncated from \(x\). We have tested this program against an oracle that captures the proposed loop function using 945 test cases; all were successful.

1.2 Premises

The goal of the research discussed in this paper is to derive the function of a while loop automatically, from a static analysis of its source code. We carry out this research under the following premises:

- **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 properties) 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, not a complete description of the loop’s functionality.

- **Evolving Capability.** The algorithm we have implemented to derive the function of a loop is made up, at its core, of two orthogonal components: A knowledge component, that contains information about program patterns and their functions; and an inference component, that matches source code against cataloged program patterns and makes inferences about the function of the loop. The latter component has been developed, and is in a reasonably final form. The first component, which determines the capability of the algorithm, evolves as we store more and more programming knowledge or domain-specific knowledge. 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 properties 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.

In the next section, we briefly introduce some mathematical concepts that we use in this paper, pertaining primarily to relational algebra and a relation-based refinement calculus. In section 3 we use the background introduced in section 2 we present a set of theorems that form the basis of our approach. In section 4 we discuss how the results of section 3 can be deployed to derive an algorithm that extracts the function of a loop; we also revisit the examples discussed above to illustrate how our algorithm applies to them. In section 5 we discuss theoretical and practical extensions to our work,
and in section 6 we discuss related work. Finally section 7 summarizes our main findings and attempts to assess their

significance.

2 Mathematical Background

We represent the functional specification of programs by relations; without loss of generality, we consider homogeneous

relations, and we denote by \( S \times S \) 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

\[
s = (x, y, z).
\]

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 \( I \), the identity relation, denoted by \( I \), and the empty relation, denoted

by \( \phi \). Given a predicate \( t \), we denote by \( I(t) \) the subset of the identity relation defined as follows:

\[
I(t) = \{(s, s') | s' = s \land t(s)\}.
\]

Because relations are sets, we use the usual set theoretic operations between relations. Operations on relations also

include the converse, denoted by \( \bar{R} \) or \( R \sim \), and defined by

\[
\bar{R} = \{(s, s') | (s', s) \in R\}.
\]

The product of relations \( R \) and \( R' \) is the relation denoted by \( R \circ R' \) (or \( RR' \)) and defined by

\[
R \circ R' = \{(s, s') | \exists t : (s, t) \in R \land (t, s') \in R'\}.
\]

The prerestiction (resp.post-restricion) of relation \( R \) to predicate \( t \) is the relation \( \{(s, s') | t(s) \land (s, s') \in R\} \) (resp.

\( \{(s, s') | (s, s') \in R \land t(s')\} \)). We admit without proof that the pre-restriction of a relation \( R \) to predicate \( t \) is \( I(t) \circ R \)

and the post-restricion of relation \( R \) to predicate \( t \) is \( R \circ I(t) \). The domain of relation \( R \) is defined as \( \text{dom}(R) = \{s | \exists s' : (s, s') \in R\} \). The range of relation \( R \) is denoted by \( \text{rng}(R) \) and defined as \( \text{dom}(R) \). The nucleus of relation \( R \) is the relation denoted by \( \mu(R) \) and defined by \( RR \). For any \( R \), the nucleus of \( R \) is symmetric and reflexive on \( \text{dom}(R) \).

We say that \( R \) is deterministic (or that it is a function) if and only if \( RR \subseteq I \), and we say that \( R \) is total if and only if

\( I \subseteq RR \), or equivalently, \( RL = L \), and surjective if and only if \( LR = L \). 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 \( R = \bar{R} \). We will

occasionally refer to Tarski’s Identity [35, 36], 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

\[
RL \cap R' \cap (R \cup R') = R'.
\]
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 \([15, 30]\). We abbreviate this property by \( R \supseteq R' \) or \( R' \subseteq R \).

The definition of refinement that we present above (Definition 1) is complicated because we represent specifications by arbitrarily partial, arbitrarily non-deterministic relations. If we had restricted ourselves to total relations (as does Hehner \([16]\)) then refinement would be synonymous with: \textit{subset}. If we had restricted ourselves to deterministic relations (as does Mills et al. \([23, 29]\)) then refinement would be synonymous with: \textit{superset}. When specifications are neither necessarily total nor necessarily deterministic, we get the formula given in definition 1.

We admit that, modulo traditional definitions of total correctness \([11, 15, 25]\), the following propositions hold.

- A program \( P \) is correct with respect to a specification \( R \) if and only if \( [P] \supseteq R \), where \([P]\) is the function defined by \( P \).
- \( R \supseteq R' \) if and only if any program correct with respect to \( R \) is correct with respect to \( R' \).

Intuitively, \( R \) refines \( R' \) if and only if \( R \) represents a stronger requirement than \( R' \). 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 \textit{meet}, denote by \( \cap \), and define by:
  \[
  R \cap R' = RL \cap R'L \cap (R \cup R').
  \]

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

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

- 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 \textit{universal lower bound}, which is the empty relation.

- The lattice of refinement admits no \textit{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:

\[
R \cup R' = R \cap R'.
\]

Then the meet of \( R \) and \( R' \) is given by the following formula:

\[
R \cap R' = R \cup R'.
\]

The condition of this proposition means that \( R \) and \( R' \) have the same domain, and for each element of their common domain, they have at least one image in common.
3 Tenets of a Stepwise Approach

3.1 Mathematical Basis

We consider a program $P$ on some variables $x_1, x_2, \ldots, x_n$ of types (respectively) $X_1, X_2, \ldots, X_n$. We let $S$ be the Cartesian product

$$S = X_1 \times X_2 \times \cdots \times X_n,$$

and we denote by $s$ an arbitrary element of $S$. The function of program $P$ is denoted by $[P]$ and defined by

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

From this definition, it stems that $\text{dom}([P])$ can be interpreted as

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

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

**Theorem 1 Sub-goal Induction Theorem.** Given a while loop $w = \text{while } t \text{ do } B$ 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:

- $\text{dom}([w]) = S$.
- $I(\neg t) \subseteq R$.
- $I(t) \circ [B] \circ R \subseteq R$.

This theorem is due to Morris and Wegbreit [31].
Theorem 2 Mills’ Theorem. Given a while loop \( w = \text{while } t \text{ do } B \) 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:
- \( \text{dom}(F) = S \).
- \( I(\neg t) \circ F = I(\neg t) \).
- \( I(t) \circ [B] \circ F = I(t) \circ F \).

This theorem is due to H.D. Mills [29]. 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 \( S \) of the form:
\[
w: \text{while } t \text{ do } B.
\]
We let \( s_0 \) be an initial state on which \( w \) is executed and terminates, and we let \( s_1, s_2, s_3, \ldots, s_n \) be the sequence of states produced by the execution of \( w \) on \( s_0 \), where \( s_n \) is the final state. We claim that for all \( i, 0 \leq i \leq n, s_i \) is in \( \text{dom}([w]) \).

Proof. Initial state \( s_0 \) is by definition in \( \text{dom}([w]) \), since execution of the loop starting at \( s_0 \) terminates. Final state \( s_n \) is also in \( \text{dom}([w]) \), since it satisfy condition \( t(s_n) \), hence the execution of the loop on \( s_n \) terminates immediately. Intermediate states \( s_i \), for \( 1 \leq i \leq n - 1 \), are also in \( \text{dom}([w]) \), since if they weren’t, the loop would not terminate for them, whence it would not terminate for the initial state \( s_0 \).

Since initial states, intermediate states, and final states are all in \( \text{dom}([w]) \), we can let \( S \) be \( \text{dom}([w]) \) without loss of generality, as then all the states of interest are within \( \text{dom}([w]) \). 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; a possible avenue for this may be the work of Berdine et al [2] on variant assertions, and their application to the termination of loops. 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 We consider a while statement of the form \( w = \text{while } t \text{ do } B \). 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: \( \text{dom}(F) = S \). We assume that there exists an element of \( s \) of \( S \) that is outside \( \text{dom}(F) \). 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 \(s\)). We infer
\[ s \in \text{dom}((I(t) \circ [B])^n I(\neg t)), \]
from which we infer, in turn,
\[ s \in \text{dom}((I(t) \circ [B])^* I(\neg t)), \]
(by definition of the reflexive transitive closure), from which we infer (by substitution),
\[ s \in \text{dom}(F), \]
which contradicts the assumption that \(s\) is outside the domain of \(F\).

Second condition:
\[
\begin{align*}
I(\neg t) \circ F \\
= & \{ \text{substitution of } F \} \\
I(\neg t) \circ (I(t) \circ [B])^* I(\neg t) \\
= & \{ \text{decomposition of the reflexive transitive closure} \} \\
I(\neg t) \circ ((I(t) \circ [B])^+ \circ I(\neg t) \cup I(\neg t)) \\
= & \{ \text{distributivity} \} \\
I(\neg t) \circ ((I(t) \circ [B])^+ \circ I(\neg t)) \cup I(\neg t)^2 \\
= & \{ \text{simplification} \} \\
I(\neg t)^2 \\
= & \{ \text{relational identity} \} \\
I(\neg t).
\end{align*}
\]

Third Condition:
\[
\begin{align*}
I(t) \circ [B] \circ F \\
= & \{ \text{Substitution} \} \\
I(t) \circ [B] \circ (I(t) \circ [B])^+ \circ I(\neg t) \\
= & \{ \text{Relational Identity} \} \\
(I(t) \circ [B])^+ \circ I(\neg t) \\
= & \{ \text{Pre-restricting a relation to a superset of its domain} \} \\
I(t) \circ ((I(t) \circ [B])^+ \circ I(\neg t)) \\
= & \{ \text{Adding a null term} \} \\
I(t) \circ (I(t) \circ [B])^+ \circ I(\neg t) \cup I(t) \circ I(\neg t) \\
= & \{ \text{Left factoring} \} \\
I(t) \circ ((I(t) \circ [B])^+ \circ I(\neg t) \cup I(t)) \\
= & \{ \text{Right factoring} \} \\
I(t) \circ ((I(t) \circ [B])^+ \cup I) \circ I(\neg t) \\
= & \{ \text{Definition of Reflexive Transitive Closure} \} \\
I(t) \circ (I(t) \circ [B])^+ \circ I(\neg t) \\
= & \{ \text{Substitution} \} \\
I(t) \circ F.
\end{align*}
\]
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. It attains the reflexive transitive closure of (i.e. the smallest reflexive transitive superset of) \( I(t) \circ [B] \) by taking the join (as we will see) of a succession of reflexive transitive supersets of \( I(t) \circ [B] \).

**Theorem 4** We consider a while loop \( w \) on space \( S \), defined by \( w = \text{while } t \text{ do } B \). 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

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

**Proof.** We let \( T \) be defined as

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

and we note that \( T \) is total, since

\[
\begin{align*}
TL &= \{ \text{substitution} \} \\
(R \cup I) \circ I(\neg t) \circ L \therefore \{ \text{monotonicity} \} \\
R \circ I(\neg t) \circ L &= \{ \text{hypothesis of the theorem} \}
\end{align*}
\]

\( L \).

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

\[
[w] \land TL \land ([w] \cup T)
\]

\( = \{ \text{Because } [W] \text{ is total} \} \\
TL \land ([w] \cup T)
\]

\( = \{ \text{Because } T \text{ is total, as shown above} \} \\
([w] \cup T)
\]

\( = \{ \text{substituting } [w] \text{ and } T \} \\
(I(t) \circ [B])^* \circ I(\neg t) \cup (R \cup I) \circ I(\neg t)
\]

\( = \{ \text{factoring, associativity} \} \\
((I(t) \circ [B])^* \cup R \cup I) \circ I(\neg t)
\]

\( = \{ \text{substitution} \} \\
((I(t) \circ [B])^* \cup R \cup I) \circ I(\neg t)
\]

\( = \{ \text{simplification} \} \\
((I(t) \circ [B])^* \cup R \cup I) \circ I(\neg t)
\]

\( = \{ \text{Because } R \text{ is transitive} \} \\
((I(t) \circ [B])^* \cup R^+ \cup I) \circ I(\neg t)
\]

\( = \{ \text{monotonicity of transitive closure, and hypothesis} \} \\
(R^+ \cup I) \circ I(\neg t)
\]

\( = \{ \text{because } R \text{ is transitive} \} 
\]
\[(R \cup I) \circ I(\neg t) = \{ \text{substitution} \} T. \]

\textbf{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 (in the refinement ordering) 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 it is 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.

**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]\supseteq (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]\supseteq 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. Deriving loop functions, just like generating loop invariants, aims to to discover (reverse engineer) 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 $R \circ I(-t)$. 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 4 and its corollaries are not constructive, in the sense that they are dependent on the derivation of some reflexive transitive relation $R$ that satisfies some conditions, but give us no guidance for deriving $R$. In the next two subsections, we consider constructive alternatives, focusing in turn on equivalence relations (which are symmetric, in addition to being reflexive and transitive) and rectangular relations (in which each element of the domain is related to each element of the range).

### 3.2 Equivalence Relations

We consider a while statement defined on space $S$ by \texttt{while t do B} and we assume that execution of $B$ increases an integer variable $x$ by a constant value $c$. Then, without knowing anything else about $B$, we argue that the following relation is a reflexive transitive superset of $B$:

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

It is clearly reflexive and transitive; furthermore, it is a superset of $[B]$, since $x' = x + c$ logically implies

$$x \mod c = x' \mod c.$$  

If we assume that the loop body also decrements some other integer variable $y$ by $c$ at each execution, then we can infer that the following relation, which is reflexive and transitive, is also a superset of $[B]$.

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

To generalize this kind of reasoning, we introduce the concept of \textit{invariant function}, which is due to Mili et al. [28].

**Definition 2** Let $w$ be the while statement \texttt{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 5** If $F$ is an invariant function for the while loop $w = \texttt{while t do B}$, and further satisfies the condition

$$F \circ \hat{F} \circ I(-t) \circ L = L$$

then $[w]$ refines the following specification

$$T = F \circ \hat{F} \circ I(-t).$$

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

1. $R$ is transitive,
2. $I(t) \circ [B] \subseteq R$,
3. $R \circ I(-t) \circ L \supseteq L$.

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

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

$$= \{ \text{associativity} \}$$

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

$$\subseteq \{ \text{$F$ is deterministic} \}$$

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

$$= \{ \text{trivial simplification} \}$$

$$F \circ \hat{F}.$$
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:

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

\[
\Leftrightarrow \quad \{ \text{because } I(t) \subseteq I \} 
I(t) \circ [B] \circ F \subseteq F
\]

\[
\Leftrightarrow \quad \{ \text{monotonicity of product} \} 
I(t) \circ [B] \circ F \subseteq F \circ \hat{F}
\]

\[
\Leftrightarrow \quad \{ \text{because } F \text{ is total, } I \subseteq F \circ \hat{F} \} 
I(t) \circ [B] \circ I \subseteq F \circ \hat{F}
\]

\[
\Leftrightarrow \quad \{ \text{trivial simplification} \} 
I(t) \circ [B] \subseteq F \circ \hat{F}
\]

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 \([u]\) refines the following specification:

\[
T = (F \circ \hat{F} \cup I) \circ I(-t).
\]

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

\[
(F \circ \hat{F} \cup I) = F \circ \hat{F}.
\]

Substituting in the formula of \( T \) above, we find the result of this theorem. \( \text{qed} \)

The importance of this theorem is that it allows us to map any invariant function that we find onto a lower bound of the loop function. Of course, now the question is: how do we derive invariant functions? Mili et al [28] had drawn a list of (strongest) invariant functions. In section 4.2 we broaden the list of Mili et al and organize it to support our extraction algorithm.

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.

### 3.3 Rectangular Relations

In the previous subsection, we have explored transitive relations which are derived as nuclei \((F \circ \hat{F})\) 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 6  Let $w$ be a while loop defined by \texttt{while } $t$ \texttt{do } B. If $t \neq \texttt{false}$ then

$$[w] \supseteq T$$

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

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

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

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

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

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

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

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 $\text{rng}([w]) = \{ s| t(s) \}$; and we know by hypothesis that $t \neq \texttt{false}$. Hence $\text{rng}([w]) \neq S$. Let $s$ be an element of $\text{rng}([w])$. Because state $s$ is outside $\text{rng}([w])$, it satisfies predicate $t$; whence application of the loop on $s$ will execute $B$; because we are assuming that

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

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 $\text{dom}([w])$. We conclude that

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

therefore (by Tarski’s identity)

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

To prove that $T$ is total, we compute $TL$: 
The proof of the fourth premise is straightforward:

\[
T L
= \{ \text{substitution} \}
I(t) \circ L \circ I(t) \circ [B] \circ I(-t) \circ L \cup I(-t) \circ L
= \{ \text{associativity} \}
I(t) \circ (L \circ I(t) \circ [B] \circ I(-t) \circ L) \cup I(-t) \circ L
= \{ \text{lemma above} \}
I(t) \circ L \cup I(-t) \circ L
= \{ \text{factorization} \}
(I(t) \cup I(-t)) \circ L
= \{ \text{simplification} \}
I \circ L
= \{ \text{identity} \}
L.
\]

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

\[
t \not= \text{false},
\]

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

\[
\text{while } i>1 \text{ do } i:= i-1
\]

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 \((i \not= 1)\).

The following theorem provides a second lower bound of the loop function in the form of a rectangular relation. It is useful whenever the function of the loop body is not surjective.
**Theorem 7** We consider a while statement $w$ of the form `while t do B` on space $S$, such that $w$ terminates for all initial states in $S$, and that $t \neq \text{false}$. Then the function of this while statement ($\left< w \right>$) 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 $\text{false}$, 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 $R$.

$$RR$$

$$= \text{ substitution }$$

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

$$= \text{ associativity }$$

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

$$= \{ \text{ [B] is not empty, Tarski’s identity } \}$$

$$L \circ [B]$$

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

$$R.$$

- **$R \circ I(\neg t) \circ L = L$.** 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 $\text{false}$, 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 $B$ after execution of loop body $B$ would always yield $\text{true}$. We infer that 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 $\text{false}$) and the pathological case where the loop never terminates (if $t$ is guaranteed to be $\text{true}$ 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:

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

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 supersets.

**Proposition 3** We consider a while statement \( w \) of the form \( \text{while } t \text{ do } B \) on space \( S \), such that \( w \) terminates for all initial states in \( S \), and that \( t \neq \text{false} \), 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:

\[
T = (R \cup I) \circ I(-t).
\]

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

\[
\begin{align*}
\text{w} &= \text{while } z<>1 \text{ do} \\
&\quad \{ x:= x+2; \ y:= y-3; \ z:= x+y \} 
\end{align*}
\]

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

\[
z = 1 \lor x + y \geq 1.
\]

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

\[
z = 1 \lor (x + y) \geq 1.
\]

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

We apply theorem 4 to this loop, using the following reflexive transitive relation \( R \) (we will see in section 4.2 how to derive \( R \) systematically),

\[
R = \{(s, s') | x \mod 2 = x' \mod 2 \land 3x + 2y = 3x' + 2y'\},
\]

whence we derive the following lower bound:

\[
T1 = \{(s, s') | x \mod 2 = x' \mod 2 \land 3x + 2y = 3x' + 2y' \land z' \leq 1\}.
\]

We now apply theorem 7 to identify another lower bound. This theorem requires that we compute the range of \([B]\).

\[
L \circ [B] = \begin{cases} \text{Interpretation of relational product} \\
\{(s, s') | \exists s'': (s, s'') \in L \land (s''', s') \in [B]\} \\
= \begin{cases} \text{simplification, since } L \text{ is universal relation} \\
\{(s, s') | \exists s''': (s''', s') \in [B]\} \\
= \begin{cases} \text{substitution of } [B] \} \\
\{(s, s') | \exists s'': x'' = x'' + 2 \land y'' = y'' - 3 \land z'' = x'' + y'' \} \\
= \begin{cases} \text{arithmetic substitution} \}
\end{cases}
\end{cases}
\end{cases}
\]
\[(s, s') \exists s'': x' = x'' + 2 \land y' = y'' - 3 \land z' = x' - 2 + y' + 3\]  
\(= \)  
\{\text{arithmetic simplification, logical simplification}\}  
\[(s, s') \exists z': x' = x' + y' + 1 \land \exists s'': x' = x'' + 2 \land y' = y'' - 3\]  
\(= \)  
\{\text{logical simplification (since } x \text{ and } y \text{ 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  
\[T2 = \{(s, s') | z' = x' + y' + 1 \land z' \leq 1\} \cup I(z \leq 1)\].

Taking the join of these two relations, we find:

\([w]\)  
\(\sqsubseteq\)  
\{\text{Lattice Property}\}  
\(T1 \sqcup T2\)  
\(= \)  
\{\text{substitution, simplification}\}  
\[(s, s') | z' \neq 1 \land x \mod 2 = x' \mod 2 \land 3x + 2y = 3x' + 2y' \land z' = x' + y' + 1 \land z' = 1\} \cup I(z = 1)\]  
\(= \)  
\{\text{merging two expressions of } z'\}  
\[(s, s') | z' \neq 1 \land x \mod 2 = x' \mod 2 \land 3x + 2y = 3x' + 2y' \land 0 = x' + y' \land z' = 1\} \cup I(z = 1)\]  
\(= \)  
\{\text{highlighting the form } (x' + y')\}  
\[(s, s') | z' \neq 1 \land x \mod 2 = x' \mod 2 \land 3x + 2y = x' + 2(x' + y') \land 0 = x' + y' \land z' = 1\} \cup I(z = 1)\]  
\(= \)  
\{\text{substitution}\}  
\[(s, s') | z' \neq 1 \land x \mod 2 = x' \mod 2 \land 3x + 2y = x' \land 0 = x' + y' \land z' = 1\} \cup I(z = 1)\]  
\(= \)  
\{\text{inferring an expression for } y'\}  
\[(s, s') | z' \neq 1 \land x \mod 2 = x' \mod 2 \land 3x + 2y = x' \land y' = -3x - 2y \land z' = 1\} \cup I(z = 1)\]  
\(= \)  
\{\text{highlighting the difference between } x \text{ and } x'\}  
\[(s, s') | z' \neq 1 \land x \mod 2 = x' \mod 2 \land 3x + 2y = x' \land y' = -3x - 2y \land z' = 1\} \cup I(z = 1)\]  
\(= \)  
\{\text{because } x \text{ and } x' \text{ are a multiple of } 2 \text{ apart, they have the same remainder by } 2\}  
\[(s, s') | z' \neq 1 \land x' = x + 2(x + y) \land y' = -3x - 2y \land z' = 1\} \cup I(z = 1)\].

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

### 4 An Algorithm for Computing Loop Functions

In this section, we use the mathematical results presented above to design and implement an algorithm that derives the function of a loop written in a programming language. The algorithm that we propose proceeds in three steps:

1. **Mapping the Loop from Its Source Form to an Internal Representation.** The internal representation is very similar to programming language notations, except for one detail: the loop body is represented by concurrent assignments rather than sequential assignments. The interest of using concurrent assignments will become clearer in the sequel. As for the interest of using a uniform internal representation, it allows us to handle any actual programming language, by mapping it to this representation.

2. **Mapping the Uniform Internal Representation into a Set of Lower Bounds.** This step is the core of our algorithm. It uses theorem 4 and its corollaries to produce lower bounds; and combines these lower bounds into a set of *Mathematica* (© Wolfram Research) equations, whose unknowns are the final values of the program variables.
3. **Solving the Mathematica equations.** The equations produced in the previous step are submitted to Mathematica, and the output provides the function of the loop or a lower bound that captures all we know about the function of the loop.

See Figure 4. The first step is relatively straightforward, and a prototype to convert sequential code into concurrent assignments has been developed and deployed for some time [6, 17, 24]. The third step is relatively straightforward, and is all delegated to Mathematica. We focus our attention in this section on step 2.

### 4.1 Deriving the Loop Function by Successive Approximations

The approach we advocate for step 2 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 arise then are:

- **First, How to Derive Individual Lower Bounds?** The answer to this question is at the core of our divide and conquer approach: Because the loop body is now written as a set of concurrent assignments statements (by virtue of step 1), it is structured (in relational terms) as an intersection. Since theorem 4 and its derivatives require that we derive supersets of the [$B$], we can do so by looking at one term of the intersection (i.e. one concurrent statement) at a time or two terms of the intersection (i.e. two concurrent statements) at a time, or three terms of the intersection (i.e. three concurrent statements) at a time, etc. Indeed, if [$B$] is the intersection of two terms, say $T_1 \cap T_2$, then any superset of $T_1$ is a fortiori a superset of [$B$]; this generalizes to any number of terms. We discuss this step in section 4.2.

- **Second, 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 $T_1, T_2, \ldots, T_k$ that are all refined by [$w$], then we can infer that

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

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. We discuss this step in section 4.3.
Third, How do we know when to stop? The simplest solution is to derive all the lower bounds that we can, in light of current extraction capability; this is what our algorithm does currently. This brute force approach is prone to produce redundant equations, and to be time-wasteful, but any attempt to control redundancy or enhance time performance would also complicate the algorithm. We discuss this matter in section 4.4.

4.2 Deriving Lower Bounds

In section 3.2 we had discussed the concept of invariant function and explored its link to lower bounds of the loop function. In this section we briefly review some simple code patterns, show their corresponding invariant functions, and associated lower bounds. The aggregate of a state definition, a code pattern, and associated lower bound is called a Semantic Recognizer, or recognizer for short. We classify recognizers into three categories: One-Recognizers, whose code pattern includes a single concurrent assignment; Two-Recognizers, whose code pattern includes two concurrent assignments; Three-Recognizers, whose code pattern includes three concurrent assignments. To keep combinatorics under control, we are limiting ourselves, for the time being, to recognizers of length 3, but this may change. Our algorithm derives lower bounds of loop functions by matching code patterns of recognizers against combinations of concurrent statements from the loop body, and composing these lower bounds to obtain the loop function or an approximation thereof.

4.2.1 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 a sample 1-Recognizer below.

<table>
<thead>
<tr>
<th>State Space</th>
<th>Code Pattern</th>
<th>Invariant Function</th>
<th>Lower Bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>x: int</td>
<td>x:=x+c</td>
<td>( x \mod c )</td>
<td>( {(s, s')</td>
</tr>
</tbody>
</table>

4.2.2 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 sample 2-Recognizers below, where First and Rest 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.

<table>
<thead>
<tr>
<th>ID</th>
<th>State Space</th>
<th>Code Pattern</th>
<th>Invariant Function</th>
<th>Lower Bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>2R1</td>
<td>x, y: int const a, b: int</td>
<td>x = x+a y = y+b</td>
<td>ay-bx</td>
<td>( T = {(s, s')</td>
</tr>
<tr>
<td>2R2</td>
<td>x, y: int const a: int</td>
<td>x = x*a y = y+x</td>
<td>( y(1-a) + x )</td>
<td>( T = {(s, s')</td>
</tr>
<tr>
<td>2R3</td>
<td>x, y: int const a, b: int</td>
<td>x = x+a y = y*b</td>
<td>( y/a \mod a )</td>
<td>( T = {(s, s')</td>
</tr>
<tr>
<td>2R4</td>
<td>x: listType y: listType</td>
<td>y:=y.First(x) x:=Rest(x)</td>
<td>y-x</td>
<td>( T = {(s, s')</td>
</tr>
<tr>
<td>2R5</td>
<td>i: int x: sometype</td>
<td>i:=i-1, x:=f(x)</td>
<td>( f^i(x) )</td>
<td>( T = {(s, s')</td>
</tr>
</tbody>
</table>

For illustration, we consider a while statement that contains the following statements (where the dot represents concatenation):

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

Application of semantic recognizer 2R4 to the first and second statements produces (after simplification) the following lower bound for \( w \):

\[
T_1 = \{(s, s')| x' = \epsilon \land y' = y.x \}
\]

where \( \epsilon \) is the empty sequence. Application of recognizer 2R5 to the second and third line produces (after simplification, using the axiomatization of lists) the following lower bound for \( w \):

\[
T_2 = \{(s, s')| x' = \epsilon \land i' = i - length(x) \},
\]

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

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

### 4.2.3 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 below:

<table>
<thead>
<tr>
<th>ID</th>
<th>State Space</th>
<th>Semantic Pattern</th>
<th>Invariant Function</th>
<th>Lower Bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>3R1:</td>
<td>i: int x: sometype y: sometype</td>
<td>i:=i-1, x:=f(x) y:=y+x</td>
<td>( x + \sum_{k=1}^{i-1} f^k(x) ) ( y + \sum_{k=1}^{i'} f^k(x') )</td>
<td>( T = {(s, s')</td>
</tr>
</tbody>
</table>

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:

\[ w = \]
\[ \text{while } (i<>0) \text{ do} \]
\[ [i:= i-1, \]
\[ x:= x-1, \]
\[ y:= y+x] \]

The recognizer provides (after ample simplification) the following lower bound for \( w \):

\[
[w] \supseteq \{(s, s')| x \geq i \land x' = x - i \land
y' = y + \frac{x(x+1)}{2} - \frac{i(i+1)}{2} \land i' = 0 \}
\]

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

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.
4.3 Composing Lower Bounds

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 4** 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

\[
[w] \supseteq T_1 \\
[w] \supseteq T_2
\]

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. \hspace{1cm} \text{qed}

In practice, this proposition means that rather than add the term \( I(\neg t) \) to each lower bound, we will generate it only once. On the other hands, the lower bounds can them be limited to the reflexive transitive part (relation \( R \) in theorem 4). This will be illustrated in section 4.5.

4.4 Sufficiency

The question of when do we stop generating lower bounds of a loop written in concurrent assignments format can be interpreted in two ways:

- When is a set of lower bounds sufficient to derive the function of a loop?
- How do we ensure that we are not generating useless (redundant) lower bounds?

The answer to the first question is provided by the following proposition.

**Proposition 5** If \([w] \supseteq T\) and \( T \) is total and deterministic then \([w] = T\).

This Proposition stems readily from the lattice-like structure of the refinement ordering, illustrated in Figure 1. In this structure, 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.

As for the question of how to control redundancy, we conjecture that we could do so as follows: we represent the loop body by a graph, whose nodes represent concurrent assignments, and we place an arc between two concurrent assignments whenever we apply a recognizer that matches these two assignments (2-recognizers link two concurrent assignments; and 3-recognizers link three pairs of concurrent assignments). Then we can stop searching for lower bounds as soon as the graph becomes connected. Be that as it may, the current algorithm generates all the lower bounds it can find, letting Mathematica deal with the resulting redundancy.

4.5 Application

We revisit the four sample loop we had presented in section 1.1 and show for each what set of equations are generated by the recognizer step. The algorithm generated these equations using a library of two 1-recognizers, twelve 2-recognizers, and three 3-recognizers.
4.5.1 The Numeric Example

We rewrite the loop of the first program we had presented in section 1.1.

```c
while (i != 0)
    { v = v + a*t;
      z = z + c*x;
      w = w + e*y;
      x = x+a;
      y = y+b;
      t = t*d;
      i = i-1;
    }
```

The recognizer step produces the following equations (to which we add line number, for easy reference):

1. Reduce[ Reduce[ {
2. Mod[x, Abs[a]]==Mod[xP, Abs[a]],
3. Mod[y, Abs[b]]==Mod[yP, Abs[b]],
4. Mod[t, Abs[Log[d,10]]]==Mod[tP, Abs[Log[d,10]]],
5. Mod[i, Abs[1]]==Mod[iP, Abs[1]],
6. i>=iP,
7. a*y-b*x==a*yP-b*xP,
8. b*x-a*y==b*xP-a*yP,
9. z-c*x*(x-a)/(2*a)==zP-c*xP*(xP-a)/(2*a),
10. t/d^(x/a)==tP/d^(xP/a),
11. a*i+1*x==a*iP+1*xP,
12. t/d^(y/b)==tP/d^(yP/b),
13. w-e*y*(y-b)/(2*b)==wP-e*yP*(yP-b)/(2*b),
14. b*i+1*y==b*iP+1*yP,
15. v+a*t/(1-d)==vP+a*tP/(1-d),
16. (iP=0),
17. Exists [ {iPP,tPP,vPP,wPP,xPP,yPP,zPP},
18. !(iPP=0) &&
19. iP==iPP-1 &&
20. wP==wPP+e*yPP &&
21. vP==vPP+a*tPP &&
22. tP==tPP+d &&
23. zP==zPP+c*xPP &&
24. yP==yPP+b &&
25. xP==xPP+a]
26. ]},
```

Lines 2-6 are generated from 1-recognizers; lines 7-15 are generated from 2-recognizers; and lines 16-26 are generated from rectangular lower bounds. As the reader may notice readily, many of these equations are redundant; we have made no effort to reduce redundancy, counting on Mathematica to deal with it. Notice for example that the equation on line 7 makes equations 2 and 3 mutually redundant (i.e. we need one or the other, but not both). Hence in general we never need more than one equation of form Mod[x, a]=Mod[xP, a]. When one of the equations has 1 as the argument of the mod (i.e. Mod[x, 1]=Mod[xP, 1]) then we need no mod equations at all, since this equation is itself intrinsically redundant (it does not provide any information on i, since i is an integer variable). Notice also that because the negation of the loop condition (given on line 16) is an equality, the Exists clause is not needed.
4.5.2 The Function Call

We present the loop:

```c
while (x>5)
{
    x = x-5;
    y = y+b;
    z = z+b*x+5*b;
    j = j-1;
    t = t+c*y-b*c;
    v = f(v);
    w = w+v;
}
```

The recognizer step produces the following equations (to which we add line number, for easy reference):

1. Reduce[ Reduce[ {  
2. Mod[x, Abs[5]]==Mod[xP, Abs[5]],  
3. Mod[y, Abs[b]]==Mod[yP, Abs[b]],  
4. Mod[j, Abs[1]]==Mod[jP, Abs[1]],  
5. j>=jP,  
6. b*x+5*y==b*xP+5*yP,  
7. z+b*x*(x+5)/(2*5)==zP+b*xP*(xP+5)/(2*5),  
8. 5*j-1*x==5*jP-1*xP,  
9. 1*x-5*j==1*xP-5*jP,  
10. b*j+1*y==b*jP+1*yP,  
11. t-c*y*(y-b)/(2*b)==tP-c*yP*(yP-b)/(2*b),  
13. w+Sum[Nest[f,v,k],{k,1,j}]==wP+Sum[Nest[f,vP,k],{k,1,jP}],  
14. (xP<5),  
15. Exists[ {jPP,tPP,vPP,wPP,xPP,yPP,zPP},  
16. !(xPP<5) &&  
17. wP==wPP+vPP &&  
18. vP==f(vPP) &&  
19. tP==tPP+c*yPP &&  
20. jP==jPP-1 &&  
21. zP==zPP+b*xPP &&  
22. yP==yPP+b &&  
23. xP==xPP-5]  
24. }],  

Lines 2-5 are generated from 1-recognizers; lines 6-12 are generated from 2-recognizers; line 13 is generated from a 3-recognizer; and lines 14-23 are generated from rectangular lower bounds. The Exists clause (lines 15-23) reduces merely to $(xP \geq 0)$, which combined with line 14 yields $(0 \leq xP < 5)$. This allows us to rewrite line 2 as:

\[ xP = x \mod 5. \]

The other numeric values can be computed in sequence using this expression of $xP$. As for variables $vP$ and $wP$, they can be derived if we observe that, in light of the inequality of line 5, line 12 can be rewritten as $vP = Nest[f, v, j-jP]$. 

26
4.5.3 Array Processing

```
while (i != N+1)
{
    x = x + a[i];
    y = y + b[j];
    i = i+1;
    j = j-1;
}
```

The recognizer step produces the following equations (to which we add line number, for easy reference):

1. Reduce[ Reduce[{
2.  Mod[i,Abs[1]]==Mod[iP,Abs[1]],
3.  i<=iP,
4.  Mod[j,Abs[1]]==Mod[jP,Abs[1]],
5.  j>=jP,
6.  a==aP,
7.  b==bP,
8.  1*i+1*j==1*iP+1*jP,
9.  x+Sum[a[k],{k,i,N}]==xP+Sum[aP[k],{k,iP,N}],
10. y+Sum[b[k],{k,1,j}]==yP+Sum[bP[k],{k,1,jP}],
11. (iP==N+1),
12. Exists[ (aPP,bPP,iPP,jPP,lPP,xPP,yPP),
13.  !(iPP==N+1) &&
14.  lP==tail(lPP) &&
15.  bP==bPP &&
16.  aP==aPP &&
17.  yP==yPP+bPP[jPP] &&
18.  xP==xPP+aPP[iPP] &&
19.  jP==jPP-1 &&
20.  iP==iPP+1]
21. }],
```

Lines 2-5 are generated from 1-recognizers; lines 6-10 are generated from 2-recognizers; and lines 11-20 are generated from rectangular lower bounds. In this case, the Exists clause is redundant, since the negation of the loop condition is an equality. Also, lines 3 and 5 are mutually redundant, since line 8 provides that the sum of $i$ and $j$ is preserved. As for the mod equations (lines 2 and 4), they are both redundant.

4.5.4 The List Example

```
while (!(x.empty()))
{
    i = i-1;
    y.push_back(x.front());
    x.pop_front();
    t = t*c;
}
```

The recognizer step produces the following equations (to which we add line number, for easy reference):

1. Reduce[ Reduce[{
2.  Mod[i,Abs[1]]==Mod[iP,Abs[1]],
3.  i<=iP,
4.  Mod[j,Abs[1]]==Mod[jP,Abs[1]],
5.  j>=jP,
6.  a==aP,
7.  b==bP,
8.  1*i+1*j==1*iP+1*jP,
9.  x+Sum[a[k],{k,i,N}]==xP+Sum[aP[k],{k,iP,N}],
10. y+Sum[b[k],{k,1,j}]==yP+Sum[bP[k],{k,1,jP}],
11. (iP==N+1),
12. Exists[ (aPP,bPP,iPP,jPP,lPP,xPP,yPP),
13.  !(iPP==N+1) &&
14.  lP==tail(lPP) &&
15.  bP==bPP &&
16.  aP==aPP &&
17.  yP==yPP+bPP[jPP] &&
18.  xP==xPP+aPP[iPP] &&
19.  jP==jPP-1 &&
20.  iP==iPP+1]
21. }],
3. \( i \leq i^P \),
4. \( \text{Mod}[t, \text{Abs}[\log(c, 10)]] \equiv \text{Mod}[t^P, \text{Abs}[\log(c, 10)]] \),
5. \( x^P = \text{Nest}[\text{Rest}, x, i^P - i] \),
6. \( i - \text{Length}[x] = i^P - \text{Length}[x^P] \),
7. \( t/c^{(i/1)} = t^P/c^{(i^P/1)} \),
8. \( \text{Join}[y, x] = \text{Join}[y^P, x^P] \),
9. \( (x^P = {} ) \),
10. \( \exists \{i^P, t^P, x^P, y^P\} \),
11. \( ! (x^P = {}) \) \& \&
12. \( t^P = t^P \times c \) \& \&
13. \( y^P = \text{Join}[y^P, \text{First}[x^P]] \) \& \&
14. \( x^P = \text{Rest}(x^P) \) \& \&
15. \( i^P = i^P + 1 \)
16. \}
17. \{i^P, t^P, x^P, y^P\}, \text{Backsubstitution} \rightarrow \text{True}

Lines 2-4 are generated from 1-recognizers; lines 5-8 are generated from 2-recognizers; and lines 9-15 are generated from rectangular lower bounds. The translation from C++ to the concurrent assignment format translates the C++ list operations into Mathematica notation, so that the recognizer step produces function names that are compatible with Mathematica terminology. Though we have not checked in detail, we suspect that equation 7 makes equations 2 and 4 mutually redundant. Since equation 2 is itself (intrinsically) redundant, equations 2 and 4 can safely be disposed of.

Equation 5 was originally written in a symmetric fashion, as \( \text{Nest}[	ext{Rest}, x, i] = \text{Nest}[	ext{Rest}, x^P, i^P] \). Because \( i \leq i^P \) (line 3), this is rewritten as in line 5, to facilitate the calculation of \( x^P \). Equation 6 provides that if we keep subtracting 1 to \( i \) and popping off elements of \( x \), the difference between \( i \) and the length of \( x \) is preserved. Equation 8 provides that deleting the front element of \( x \) and attaching to the back of \( y \) preserves the list formed by concatenating \( y \) to \( x \). The \( \exists \) clause in this example does not provide additional information, as line 9 gives all the information we need about \( x^P \).

5 Extensions

In this section we briefly discuss extensions of our current work, considering in turn theoretical extensions, then practical extensions.

5.1 Theoretical Extensions

The most pressing theoretical extension is the derivation of loop functions in the presence of if-then-else constructs in the loop body. Of course, theorem 4 applies equally well whether the loop body does or does not have if-then-else constructs. However, the divide-and-conquer approach that we have implemented in our algorithm is based on the assumption that we can compute a superset of the function of the loop body by considering a subset of the concurrent assignments that define the loop body. This is only possible because each concurrent assignment represents a term of an intersection: If the loop body function is written as

\[ [B] = CA_1 \cap CA_2 \cap CA_3 \cap \ldots \cap CA_n \]

then a superset of any \( CA_i \) (or the superset of any combination \( CA_i \cap CA_j \) or \( CA_i \cap CA_j \cap CA_k \)) is, a fortiori, a superset of \([B]\). The introduction of if-then-else constructs breaks this uniform structure because it introduces union operators in the mix.

The analysis of this problem in the general case (with arbitrarily nested if then else’s) is beyond the scope of this paper. In this section we will merely present the venue we are pursuing to address this problem, and why we believe it is a relevant/fruitful research direction. We illustrate our approach on a simple example: We consider the following problem on integer variables \( x, y, z \):

28
while !(y==0)
{
  if (y%2 == 0)
    {x = 2*x; y = y/2; }
  else
    {y = y-1; z = z+x; }
}

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

\[
P = \{(s, s') | \mod 2 = 1 \land x' = x \land y' = y - 1 \land z' = z + x\},
\]

\[
Q = \{(s, s') | \mod 2 = 0 \land x' = 2 \times x \land y' = y / 2 \land z' = z\}.
\]

As we recall (Mills' Theorem, section 2), the function of the loop can be written as

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

Hence the key to deriving the function of \([w]\) (in the presence of if then else statements) is to analyze the formula that gives the transitive closure of a union of functions. In relational algebra, the transitive closure of a union is given by the following formula:

\[
(P \cup Q)^* = P^* (QP^*)^* = (P^* Q)^* P^*.
\]

By analogy with our discussions in section 3, we conjecture that if we find a reflexive transitive relation \( R \) that is a superset of \( P \) and a reflexive transitive relation \( R' \) that is a superset of \( Q \), then

\[
R \circ R' \circ I(-t)
\]

is a lower bound of \([w]\). We apply this conjecture to the example at hand; to this effect, we consider the else clause, which we rewrite as the set of concurrent assignments:

\[
x = x,
\]

\[
y = y-1,
\]

\[
z = z+x
\]

Applying 2-recognizer 2R1 (strictly speaking \( x \) ought to be a constant, but we can apply it since \( x \) is preserved), we find the following invariant function \( x \times y - (-1) \times z \), which we simplify to \( z + x \times y \). We derive the following reflexive transitive relation that is a superset of \( P \):

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

The reader can easily verify that \( R \) is reflexive and transitive, and that it is a superset of \( P \). Now, we consider \( QR \):

\[
QR = \{(s, s') | \exists t : (s, t) \in Q \land (t, s') \in R\}
\]

\[
= \{(s, s') | \exists t : y(s) \mod 2 = 0 \land x(t) = 2 \times x(s) \land y(t) = y(s) / 2 \land z(t) = z(s) \land z(t) + x(t) \times y(t) = z(s') + x(s') \times y(s')\}
\]

\[
= \{(s, s') | \exists t : y(s) \mod 2 = 0 \land z(s) + 2 \times x(s) \times (y(s) / 2) = z(s') + x(s') \times y(s')\}
\]

\[
= \{(s, s') | y(s) \mod 2 = 0 \land z(s) + x(s) \times y(s) = z(s') + x(s') \times y(s')\}.
\]
For $R'$, we take the following superset of $QR$:

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

According to our conjecture, $R \circ R' \circ I(-t)$ is a lower bound for $[w]$. We compute it as follows:

\[
\begin{align*}
T &= \{ \text{substitution} \} \\
R \circ R' \circ I(-t) &= \{ \text{since } R = R' \} \\
R^2 \circ I(-t) &= \{ R \text{ is reflexive and transitive} \} \\
R \circ I(-t) &= \{ \text{substitution} \} \\
&= \{(s, s')|z + x \times y = z' + x' \times y' \land y' = 0\} \\
&= \{ \text{simplification} \} \\
&= \{(s, s')|z' = z + x \times y \land y' = 0\}.
\end{align*}
\]

This is indeed a lower bound of $[w]$, as the reader can easily verify. Hence the venue we are exploring (which consists in using the formulas of transitive closure of a union to capture the semantics of if-then-else within a loop body) appears to bear fruit, and will be pursued further.

5.2 Practical Extensions

The most stringent bottleneck we are experiencing currently as we deploy our algorithm is the inability of Mathematica to handle some equations which we expected it to handle quite easily (given how powerful it is in other application domains). For example, Mathematica does have a notation to represent multiple applications of a function to an argument, but is unable to reason about it; specifically, if we write

\[
i \geq i \land \operatorname{Nest}[f, x, i] = \operatorname{Nest}[f, x, i].
\]

Mathematica cannot infer that $xP$ can be derived as

\[
xP = \operatorname{Nest}[f, x, i - iP].
\]

We have solved this temporarily by generating directly the second equation $xP = \operatorname{Nest}[f, x, i - iP]$ (rather than the symmetric form generated from invariant functions), facilitating Mathematica’s subsequent transformation. But by and large, we are interested in having the ability to perform this type of transformation automatically. To this effect, we are experimenting with other systems for symbolic computation, at the same time as we continue generating code for Mathematica and exploring its capabilities in greater depth.

Medium term practical extensions consist in populating the database of recognizers with new recognizers, and assessing its capability against larger and more complex programs, most notably programs that involve non trivial data types and their associated operations. Longer term practical extensions include the elimination of redundant equations, but this is hardly a bottleneck at the moment, given the small number of recognizers that we are using.

6 Related Work

Generally, the derivation of loop invariants is closely related to the derivation of loop functions since they both aim to discover the inductive argument that underlies the behavior of the loop. Furthermore, a theorem by Mills [29] shows how loop functions can be used to produce loop invariants. Also, the generation of lower bounds that we carry out to approximate the function of a loop is reminiscent of the extensive work that has been done and is being done on
generating loop invariants [19]. In [27], we discuss the difference between traditional loop invariants (in the sense of Hoare’s logic [11, 15, 18]) and the loop invariants that we derive in this paper from invariant functions, which we call reflexive transitive loop invariants. Rather than discuss all the minute differences that exist between reflexive transitive loop invariants and traditional loop invariants (which we did in [27]), we will, in this section, give formula-based characterizations of these two quantities. To this effect, we consider a while loop of the form

\[ w : \text{while } t \text{ do } B. \]

Also, we consider an initialized version of this loop, of the form

\[ f : \text{init; while } t \text{ do } B. \]

In [27] we have found that the strongest reflexive transitive loop invariant (one that is strong enough to generate the function of the loop) is given by the following formula (where \( W \) is the function of the loop):

\[(s, s') \in W \hat{W}.\]

Also, we have found in [27] that the strongest loop invariant (in the sense of Hoare), i.e. a loop invariant that is strong enough to prove the statement

\[ f : \{s=s0\} \text{init; while } t \text{ do } B \{s=F(s0)\} \]

is given by the following formula,

\[(s_0, s) \in F \hat{F} \cap L(F \hat{F} \cap W),\]

where \( F \) is the function of the initialized while loop. As an illustrative example, we briefly consider a simple example, of a loop that adds the elements of an array:

\[ f : \{x=0; i=1; \text{w: while !(i==N+1) do } (x=x+a[i]; i=i+1); \}\]

The function of program \( f \), which we denote by \( F \), is given by the following formula:

\[ F \left( \begin{array}{c} x \\ i \\ a \end{array} \right) = \left( \begin{array}{c} \sum_{k=1}^{N} a[k] \\ N+1 \\ a \end{array} \right) \]

As for the function of the loop, which we denote by \( W \), it is defined by

\[ W \left( \begin{array}{c} x \\ i \\ a \end{array} \right) = \left( \begin{array}{c} x + \sum_{k=1}^{N} a[k] \\ N+1 \\ a \end{array} \right) \]

From these definitions, we derive the following loop invariant:

\[(s_0, s) \in F \hat{F} \cap L(F \hat{F} \cap W) \]

\[ \iff \]

\[ a = a_0 \land x = \sum_{k=1}^{i-1} a[k]. \]

As for the reflexive transitive loop invariant, it is obtained simply by taking the nucleus of relation \( W \), which is:

\[(s, s') \in W \hat{W} \]

\[ \iff \]

31
As one can see from even this simple example, knowing the strongest loop invariant (in the sense of Hoare’s logic) does not necessarily help us to derive the reflexive transitive loop invariant. Nevertheless, we are exploring means to use the extensive work that is done on generating loop invariants to see if it can help us generate recognizers.

Many researchers in the theorem proving and the program verification communities have lent much attention to the goal of extracting loop invariants [4, 5, 7–9, 20–22, 26, 32–34]. In [13] 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 [10], 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 [32]. In [21, 22] 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 [20], Cousot and Halwachs [9], and Mili et al [28]. 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, 14]. The closest work we have found to our effort, in terms of goal (generating loop functions) and means (using Mills-like functional/relational logic) is work by Dunlop and Basili [12]. 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.

7 Conclusion

7.1 Summary

The goal of automatically computing the function of a while loop is in general difficult to attain, but it is a worthwhile goal, for it has a great impact on many software engineering activities, such as analysis, verification, maintenance, evolution, certification etc. In this paper we present a refinement-based approach to automatically computing the function of a while loop, which consists primarily in analyzing the source code of the loop to generate equations whose solution yields the function of the loop. This paper presents in turn:

- The mathematical results that enable us to derive the function of a while loop by a stepwise analysis of its source code.
- An algorithm that uses the mathematical results to extract the function of a loop, or an approximation thereof.
- An illustration of the algorithm on simple but diverse examples, along with an outline of how the current capability can extended to cover more loops, and more of each loop.

7.2 Assessment

We submit that the approach advocated in this paper is worthy of further consideration in light of the following premises:

- It offers a systematic approach to the derivation of loop functions, that proceeds in a stepwise fashion, by considering arbitrarily small portions of code (once the loop is translated into concurrent assignment format).
Most of the functionality is language independent, hence making it possible to handle loops in a wide range of languages (C, C++, Java), with very little overhead.

It offers virtually unlimited capacity for further extension, in the sense that we can handle any loop (without conditionals, for the time being) provided we store relevant recognizers for its data types and its operations. Of course, we anticipate that as the database of recognizers grows, performance may become an issue, but this has not arisen yet; also we could envision a situation where distinct databases of recognizers are invoked for distinct application domains, hence controlling performance degradation.

The proposed algorithm can offer some results as soon as it is able to match any code pattern in its recognizer library against the source code; this means that the algorithm can be useful even when it cannot compute the function of the loop in all its detail.

The approach that we propose to handle conditionals in the loop body appears to be promising, even though we anticipate that it may complicate the matching algorithm.

Research on this algorithm and the theory behind it is under way, as discussed in section 5.

References


