Reflexive Transitive Loop Invariants: A Basis for Computing Loop Functions

Ali Mili

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

Abstract. The search for loop invariants in the form of inductive assertions is useful to prove the correctness of loops in Hoare's logic, but is only indirectly useful to derive the function of a loop. In this paper we introduce a class of binary loop invariants, and discuss their application to the derivation of loop functions.

Keywords

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

1 Unary and Binary Loop Invariants

Loop invariants in the form of inductive assertions have been the subject of extensive research since their introduction by C.A.R. Hoare in [13]. Their primary function is to prove the correctness of a while loop with respect to a specification that takes the form of a precondition and postcondition. In this paper, we present a different type of loop invariant, and discuss its use in the derivation of loop functions.

1.1 An Illustrative Example

To give the reader some intuition on the difference between traditional inductive assertions and the proposed loop invariants, we present a simple example involving a loop that computes the sum of an array.

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

begin
x:= 0;  i:= 1;
while (i <> N+1) do
```

```
    begin
    x:= x+a[i];
    i:= i+1
    end
end
```

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

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

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

An adequate invariant for this specification is:

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

According to [13], in order to prove that the while statement is partially correct with respect to the specification $(\phi(s_0, s), \psi(s_0, s))$, it suffices to prove the following premises.

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

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

While traditional loop invariants capture a property that holds initially, and after an arbitrary number of executions of the loop body, the type of loop invariants we illustrate in this section capture a reflexive transitive relation that exists between two states s and s' that are separated by an arbitrary number of iterations (i.e. s' is obtained from s by application of an arbitrary number of loop body instances, assuming the loop condition returns true whenever is tested). In other words, whereas traditional loop invariants are inductive in terms of the current state, the loop invariants we introduce in this paper are doubly inductive, providing for two states, s and s', to vary, where s precedes s' (by an arbitrary number of iterations). Also, whereas traditional loop invariants are dependent upon the loop as well as its context (in terms of initial state, precondition/postcondition, etc) the new loop invariants depend exclusively on the loop, and remain unchanged regardless of the context in which the loop is embedded.

For illustration, we consider the following reflexive transitive relation

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

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

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

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

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

To this effect, we reinterpret the second premise,

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

in the following terms, where $[B]$ is the function of B:

$$\chi(s_0, s) \wedge t(s) \Rightarrow \chi(s_0, [B](s)).$$

We proceed by successive implications.

$$\begin{array}{ll}
 & \chi(s_0, s) \wedge t \\
\Rightarrow & \qquad \{ \text{ substitution of } \chi \ \} \\
 & (s_0, s) \in R \wedge t \\
\Rightarrow & \qquad \{ \text{ substitution of } R \ \} \\
 & a_0 = a \wedge \\
 & x_0 + \sum_{k=i_0}^{N} a_0[k] = x + \sum_{k=i}^{N} a[k] \wedge \\
 & i \neq N+1 \\
\Rightarrow & \qquad \{ \text{ Shifting one term of the sum into } x \ \} \\
 & a_0 = a \wedge \\
 & x_0 + \sum_{k=i_0}^{N} a_0[k] = x + a[i] + \sum_{k=i+1}^{N} a[k] \wedge \\
 & i \neq N+1 \\
\Rightarrow & \qquad \{ \text{ deleting unnecessary conjunct } \} \\
 & a_0 = a \wedge \\
 & x_0 + \sum_{k=i_0}^{N} a_0[k] = x + a[i] + \sum_{k=i+1}^{N} a[k] \\
\Rightarrow & \qquad \{ \text{ substitution: } \begin{pmatrix} a' \\ x' \\ i' \end{pmatrix} = [B] \begin{pmatrix} a \\ x \\ i \end{pmatrix} \ \} \\
 & a_0 = a' \wedge \\
 & x_0 + \sum_{k=i_0}^{N} a_0[k] = x' + \sum_{k=i'}^{N} a[k] \wedge \\
\Rightarrow & \qquad \{ \text{ reverse substitution of } R \ \}
\end{array}$$

$$
\begin{array}{ll}
 & (s_0, s') \in R \\
\Rightarrow & \qquad \{ \text{ reverse substitution of } [B] \text{ } \} \\
 & (s_0, [B](s)) \in R,
\end{array}
$$

which is what we wanted to prove.

Of course, we did not introduce reflexive transitive loop invariants just to derive traditional invariant assertions; rather we introduced them because they allow us to generate the function of while loops, as we will discuss later in the paper. But we resolved to prove the formula

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

only to justify that we can refer to R as a loop invariant, in the traditional sense.

1.2 Narrative Characterizations

In this subsection, we build on the intuition conveyed by the example above about the contrast between traditional loop invariants and the new form of loop invariants to elucidate the various dimensions of contrast between them, in preparation for giving a formal definition and formal characterizations of these concepts. Specifically, we list the following premises:

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

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

  but we did not check the initial condition

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

  nor the final condition

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

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

- *Different Levels of Generality.* Specifically, we argue that binary loop invariants subsume unary loop invariants, in the following sense: from any binary loop invariant (R), we can generate a unary loop invariant (χ) that satisfies Hoare's invariance condition

$$\{\chi() \wedge t\}\ B\ \{\chi()\}.$$

- *Different Methods.* Due to their unary nature, unary loop invariants can be derived and analyzed by induction on the trace of execution of the loop: if the invariant holds up to a point in the execution trace, then it holds after one more execution of the loop body. Because binary loop invariants involve two arguments (s and s'), a simple induction on the execution trace does not do them justice; then we recourse to an induction on the loop structure (if the function of the loop body satisfies this property, then the function of the loop satisfies that property).

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

In the next section, we introduce some mathematical background, which we use in section 3 to formally define, using relational abstractions, traditional loop invariants and the new form of loop invariants. In section 4 we discuss characterizations of (strongest) unary loop invariants and binary loop invariants, to elucidate their subtle relations. Finally we discuss the application of binary loop invariants in section 5 and conclude with a summary and assessment of our findings as well as a review of some related work.

2 Mathematical Background

2.1 Elements of Relations

We represent the functional specification of programs by relations; without much loss of generality, we consider homogeneous relations, and we denote by S the space on which relations are defined. A relation R on set S is a subset of the Cartesian product $S \times S$, hence it is natural to represent general relations as

$$R = \{(s, s')|p(s, s')\},$$

for some binary predicate p. Typically, set S is defined by some variables, say x, y, z; hence 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 (resp. 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).

Constant relations include the *universal* relation, denoted by L, the *identity* relation, denoted by I, and the *empty* relation, denoted by $\emptyset$. 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 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* [25, 26], which provides that for any relation R, $LRL = L$ if and only if R is non-empty.

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

2.2 Relations Based Refinement

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

Definition 1. *A relation R is said to* refine *a relation R' if and only if*

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

In set theoretic terms, this equation means that the domain of R is a superset of (or equal to) the domain of R', and that for elements in the domain of R', the set of images by R is a subset of (or equal to) the set of images by R'. This is similar, of course, to refining a pre/postcondition specification by weakening its precondition and/or strengthening its postcondition [12, 22]. We abbreviate this property by $R \sqsupseteq R'$ or $R' \sqsubseteq R$. We admit that, modulo traditional definitions of total correctness [9, 12, 17], 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] Boudriga et al. analyze the lattice properties of this ordering and find the following results:

- Any two relations R and R' have a greatest lower bound, which we refer to as the *meet*, denote by $\sqcap$, and define by:

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

- Two relations R and R' have a least upper bound if and only if they satisfy the following condition:

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

  Under this condition, their least upper bound is referred to as the *join*, denoted by $\sqcup$, and defined by:

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

- Two relations R and R' have a least upper bound if and only if they have an upper bound; this property holds in general for lattices, but because the refinement ordering is not a lattice (since the existence of the join is conditional), it bears checking for this ordering specifically.
- The lattice of refinement admits a *universal lower bound*, which is the empty relation.
- The lattice of refinement admits no *universal upper bound*.
- Maximal elements of this lattice are total deterministic relations.

See Figure 1.

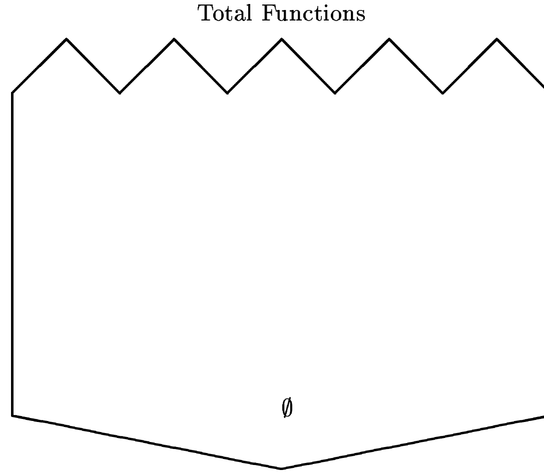

Fig. 1. Lattice Structure of Refinement

3 Relational Definitions

Because binary loop invariants involve two states, it is natural to represent them with (binary) relations; for the sake of uniformity, we will also use binary relations (specifically, vectors) to represent unary loop invariants. In this paper we assume implicitly that loops terminate for all elements of their space, i.e. they define total functions; we have shown in [20] that this hypothesis does not restrict generality (we can always redefine the space of the loop to be the set of states on which the loop terminates —Mili et al. [20] show that this does not exclude any states of interest).

3.1 Unary Loop Invariants

We consider a while loop of the form

```
w =
while t do B
```

on space S, and we let $\chi()$ be a loop invariant of w. The invariance condition of $\chi()$ can be written as

$$\{\chi() \wedge t\}\ B\ \{\chi()\}.$$

We interpret this condition in logical terms as:

$$\forall s, s' : \chi(s) \wedge t(s) \wedge (s, s') \in B \Rightarrow \chi(s').$$

If we let V be the vector defined by

$$V = \{(s, s')|\chi(s)\}$$

and T be the vector defined by

$$T = \{(s, s')|t(s)\},$$

then we can rewrite this condition as:

$$\forall s, s' : (s, s') \in V \cap T \cap B \Rightarrow (s, s') \in \widehat{V}.$$

Given that s and s' are arbitrary, this can be written algebraically as,

$$V \cap T \cap B \subseteq \widehat{V}.$$

Whence the following definition,

Definition 2. *Given a while statement of the form,* `w = while t do B`*, a unary loop invariant is defined as a vector V on S that satisfies the following condition:*

$$V \cap T \cap B \subseteq \widehat{V},$$

where T is the vector defined by predicate t.

3.2 Binary Loop Invariants

Because binary loop invariants involve two states, a past state and a current state, it is natural to represent them with binary relations.

Definition 3. *Given a while loop of the form* `w = while t do B` *on some space S, and given a relation R on S, we say that R is a* binary loop invariant *for w if and only if R is reflexive, transitive, and satisfies the following conditions (where T is the vector defined by predicate t):*

- *The Invariance Condition:*
$$T \cap [B] \subseteq R.$$
- *The Convergence condition:*
$$R \circ \overline{T} = L.$$

The following proposition explains why we refer to the first condition of definition 3 as the *invariance condition.*

Proposition 1. *Given a while statement* `w = while t do B` *on space S, and given a binary loop invariant R of w. If we let $\chi(s_0, s)$ be the predicate defined by:*

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

for some state s_0 of S, then χ satisfies the following Hoare formula:

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

Proof. By hypothesis, we have

$$T \cap [B] \subseteq R.$$

Left multiplying by R on both sides, we obtain

$$R \circ (T \cap [B]) \subseteq R \circ R.$$

Because R is transitive, we get:

$$R \circ (T \cap [B]) \subseteq R.$$

By set theory, we get:

$$(s_0, s) \in (R \circ (T \cap [B])) \Rightarrow (s_0, s) \in R.$$

By definition of the relational product, we get:

$$(s_0, s') \in R \wedge t(s') \wedge (s', s) \in [B] \Rightarrow (s_0, s) \in R.$$

Because $[B]$ is a function, we can write this as:

$$(s_0, s') \in R \wedge t(s') \Rightarrow (s_0, [B](s')) \in R.$$

Using the definition of predicate $\chi(s_0, s)$, we find

$$\chi(s_0, s') \wedge t(s') \Rightarrow \chi(s_0, [B](s')).$$

Interpreting this in the Hoare notation, and replacing the mute variable s' by the equally mute s, we find

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

qed

In other words, the invariance condition (in the sense of definition 3) of binary loop invariants yields the invariance condition (in the traditional sense of Hoare's method) of unary loop invariants.

As for the convergence condition of definition 3, it can be interpreted as follows: for any state s in space S, there exists a state s' such that (s, s') is an element of R and s' satisfies $\neg t$. In other words, R links any state s into a state s' that satisfies the termination condition $(\neg t)$.

4 Relational Characterizations

Whereas in the previous section we presented definitions of unary and binary loop invariants, in this section we attempt to give general characterizations of these invariants. In order to make the contrast between unary and binary loop invariants palatable, we aim to derive the strongest (in a sense to be defined) loop invariants of each type.

4.1 Unary Loop Invariants

We consider a while loop on space S, of the form

```
w =
while t do B
```

and we are interested in a strongest unary loop invariant for this loop. As we have discussed in section 1.2, a unary loop invariant does not depend exclusively on the loop, but on the loop's context. Hence we embed this loop in a larger program structure, which we use to derive a unary loop invariant. Specifically, we consider the following program structure, which we annotate by intermediate assertions and inductive assertions:

```
f =
begin
{s=s0}
init;
{s=[init](s0)}
while t do
   {F(s)=F(s0) &&
    s in dom(W inter F)}
    B
{s=F(s0)}
end.
```

A theorem by Mili et al. [19], which is based on earlier findings by Morris and Wegbreit [23], Mills [21] and Basu and Misra [2] provide that program f computes some total function F on S if:

1. The specification N defined by
$$N = F\widehat{F} \cap L(\widehat{F \cap W})$$
(where W is the function of the while loop) is total.
2. Segment `init` is correct with respect to specification
$$N = F\widehat{F} \cap L(\widehat{F \cap W}).$$
3. The following predicate is a loop invariant:
$$\chi(s_0, s) \equiv (s_0, s) \in F\widehat{F} \cap L(\widehat{F \cap W}).$$

To illustrate this result, we consider again the program of array sum that we used in section 1.1. The space of interest is defined by the following variable declarations:

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

As for the program structure, it is defined as follows:

```
f =
begin
x:= 0;  i:= 1;
while (i <> N+1) do
   begin
   x:= x+a[i];
   i:= i+1
   end
end
```

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

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

As for the function of the loop, it is defined by

$$W\begin{pmatrix} x \\ i \\ a \end{pmatrix} = \begin{pmatrix} x + \sum_{k=i}^{N} a[k] \\ N+1 \\ a \end{pmatrix}$$

From these definitions, we derive the specification N according to the formula provided above:

$$\begin{aligned}
& N = \\
= & \qquad \{ \text{ proposed formula } \} \\
& F\widehat{F} \cap L((\widehat{F \cap W})) \\
= & \qquad \{ \text{ substituting the first term } \} \\
& \{(s,s')|F(s) = F(s')\} \cap L((\widehat{F \cap W})) \\
= & \qquad \{ \text{ substitution, using the formula of } F \text{ } \} \\
& \{(s,s')| \textstyle\sum_{k=1}^{N} a[k] = \sum_{k=1}^{N} a'[k] \wedge \\
& N+1 = N+1 \wedge a = a'\} \\
& \cap L((\widehat{F \cap W})) \\
= & \qquad \{ \text{ logical simplifications } \} \\
& \{(s,s')|a = a'\} \cap L((\widehat{F \cap W})).
\end{aligned}$$

To continue this construction, we need to compute the intersection of F and W. We write,

$$\begin{aligned}
& F \cap W \\
= & \qquad \{ \text{ substitutions } \} \\
& \{(s,s')|x' = \textstyle\sum_{k=1}^{N} a[k] \wedge i' = N+1 \wedge a' = a \wedge
\end{aligned}$$

$x' = x + \sum_{k=i}^{N} a[k] \wedge i' = N+1 \wedge a' = a\}$

$=$ { logical simplifications }

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

$x' = \sum_{k=1}^{N} a[k] \wedge i' = N+1 \wedge a' = a\}$

$=$ { arithmetic simplification }

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

$x' = \sum_{k=1}^{N} a[k] \wedge i' = N+1 \wedge a' = a\}.$

From this we infer

$(F \cap W)L$

$=$ { domain of $(F \cap W)$ }

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

Whence the term $L(\widehat{(F \cap W)})$, which is the inverse of $(F \cap W)L$, can be written as

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

Returning to specification N, we find:

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

This specification is total; also, the initialization segment of the program of interest is correct with respect to N, since by setting x to zero and setting i to 1 it makes the following equation hold:

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

A (strongest) unary loop invariant (strong enough to capture all the functional details of F) for this while loop is then

$$(s_0, s) \in N,$$

which we interpret as

$$a = a_0 \wedge x = \sum_{k=1}^{i-1} a[k],$$

which is the same loop invariant that we had proposed in section 1.1, only there we proposed it intuitively, whereas here we compute it from our formula.

4.2 Binary Loop Invariants

Whereas unary loop invariants depended on the loop as well as its context, binary loop invariants depend solely on the while loop. The following proposition provides a strongest binary loop invariant for a while loop.

Proposition 2. *Given a while loop* `w = while t do B` *on some space S, we let W be the function of w. Then, the relation $R = W\widehat{W}$ is a binary loop invariant for w.*

Proof. We verify in turn all four conditions for binary loop invariants.

- *Reflexivity.* Generally, $W\widehat{W}$ is a superset of $I(dom(W))$. Because W is total, $I(dom(W)) = I$.
- *Transitivity.* We must prove that $RR \subseteq R$. We write

$$
\begin{aligned}
& RR \\
= & \qquad \{ \text{ substitution } \} \\
& (W\widehat{W})(W\widehat{W}) \\
= & \qquad \{ \text{ associativity } \} \\
& W(\widehat{W}W)\widehat{W}) \\
\subseteq & \qquad \{ \text{ determinacy of } W\text{, monotonicity } \} \\
& W(I)\widehat{W}) \\
= & \qquad \{ \text{ simplification } \} \\
& W\widehat{W} \\
= & \qquad \{ \text{ substitution } \} \\
& R.
\end{aligned}
$$

- *Invariance Condition.* A theorem by Mills [21] provides that if W is the function of the loop, then the following condition holds:

$$t(s) \Rightarrow W(s) = W([B](s)).$$

  We analyze this premise as follows:

$$
\begin{aligned}
& t(s) \Rightarrow W(s) = W([B](s)) \\
\Rightarrow & \qquad \{ \text{ rewriting } \} \\
& t(s) \wedge s' = [B](s) \Rightarrow W(s) = W(s') \\
\Leftrightarrow & \qquad \{ \text{ Relational rewriting } \} \\
& (s, s') \in T \cap [B] \Rightarrow (s, s') \in W\widehat{W} \\
\Rightarrow & \qquad \{ \text{ since } s \text{ and } s' \text{ are arbitrary } \} \\
& T \cap [B] \subseteq W\widehat{W}
\end{aligned}
$$

- *Convergence condition.* For all s such that $\neg t(s)$, we know from [21] that $W(s) = s$. We infer from this that $I(\neg t) \subseteq W$. We use this corollary in the following proof. We write,

$$W\widehat{W} \circ \overline{T}$$

$\supseteq$ { corollary above }

$$W \circ I(\neg t) \circ \overline{T}$$

$=$ { rewriting the vector $\overline{T}$ }

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

$=$ { idempotence }

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

$=$ { postrestricting a relation to its range }

$$W \circ L$$

$=$ { W is total }

$$L.$$

qed

Proposition 2 shows that $W\widehat{W}$ is a binary loop invariant, but does not show that it is a strongest binary loop invariant, not to mention that we did not even define what it means to be strongest. The next section will elucidate both questions.

5 Binary Loop Invariants and Loop Functions

5.1 Theorem of Binary Loop Invariants

The interest of binary loop invariants is reflected in the following theorem, due to [20].

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

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

where T is the vector defined by predicate t.

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

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

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

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

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

- Rather than ask us to derive the smallest superset of $(T \cap [B])$ that is reflexive and transitive, it asks us for a binary loop invariant of w, which is an arbitrarily large superset of $(T \cap [B])$ that is reflexive and transitive.

– On the other hand, rather than provide us with the exact function of the loop, it provides us with a lower bound (in the refinement ordering) of the loop function.

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

$$W \sqsupseteq V_1,$$

$$W \sqsupseteq V_2,$$

then we can infer

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

This approach is all the more attractive if we write the loop body B as a set of concurrent assignments, because then we can can derive supersets of B by looking at any subset of the concurrent assignments at a time (for example, one at a time, two at a time, three at a time).

5.2 A Numeric Analogy

To illustrate what it means to resolve an equation in a lattice by means of inequalities, we briefly present a numeric analogy, using a lattice whose structure is very similar to the lattice 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.

5.3 Deriving Lower Bounds

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

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

5.4 Sample 1-Recognizer

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

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

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

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

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

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

Fig. 2. Sample 1-Recognizer

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

Fig. 3. Sample 2-Recognizer

5.5 Sample 2-Recognizers

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

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

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

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

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

Fig. 4. Sample 3-Recognizer

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

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

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

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

5.6 Sample 3-Recognizer

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

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

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

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

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

6 Conclusion

6.1 Summary and Assessment

In this paper, we have briefly presented the concept of *binary loop invariant* and contrasted it to the more commonly used *unary loop invariant.* Also, we have briefly and cursorily discussed how binary loop invariants can be used to derive

unary loop invariants, and most importantly how they can be used to derive loop functions.

One of the most interesting results of this paper is perhaps the distinct characterizations we have given of a strongest (in the sense: strong enough to prove that the program computes its function) unary loop invariant and a strongest (in the sense: strong enough to derive the loop function) binary loop invariant. We have found a strongest unary loop invariant to have the form

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

where W is the function of the unitialized while loop and F is the function of the initialized while loop. On the other hand, we have found that a strongest binary loop invariant has the form

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

where W is the function of the loop. Of course these formulas are not useful for deriving these invariants but they are useful for elucidating what these invariants represent.

Another important result is that binary loop invariants can be used to derive loop functions, and that they in turn can be derived using pattern matching of data structures and control structures of the loop.

6.2 Relation to Other Work

In this section we briefly mention some samples of current research on loop invariants, and characterize their approach. In [10] Ernst et al. discuss a system for dynamic detection of likely invariants; this system, called Daikon, runs candidate programs and observes their behavior at user-selected points, and reports properties that were true over the observed executions, using machine learning techniques. Because these are empirical observations, the system produces probabilistic claims of invariance. In [8], Denney and Fischer analyze generated code against safety properties, for the purpose of certifying the code. To this effect, they proceed by matching the generated code against known idioms of the code generator, which they parameterize with relevant safety properties. Safety properties are formulated by invariants (including loop invariants), which are inferred by propagation through the code. In [6], Colon et al. consider loop invariants of numeric programs as linear expressions and derive the coefficients of the linear expressions by solving a set of linear equations; they extend this work to non linear expressions in [24]. In [15, 16] Kovacs and Jebelean derive loop invariants by solving recurrence relations; they pose the loop invariants as solutions to recurrence relations, and derive closed forms of the solution using a theorem prover (Theorema) to support the process. In [4] Rodriguez Carbonnell et al. derive loop invariants by forward propagation and fixed point computation, with robust theorem proving support; they represent loop bodies as conditional concurrent assignments, hence their insights are of interest to us as we envision

to integrate conditionals into our concurrent assignments. Less recent work on loop invariants includes work by Cheatham and Townley [5], Karr [14], Cousot and Halbwachs [7], and Mili et al. [18]. Work on loop analysis and loop transformations in the context of compiler construction is also related to functional extraction, although to a lesser degree than work on loop invariants [1, 11].

References

1. U. Banerjee. *Loop Transformations for Restructuring Compilers.* Kluwer Academic Publishers, Boston, MA, 1993.
2. S. Basu and J. Misra. Proving loop programs. *IEEE Transactions on Software Engineering*, 1(1):76–86, 1975.
3. N. Boudriga, F. Elloumi, and A. Mili. The lattice of specifications: Applications to a specification methodology. *Formal Aspects of Computing*, 4:544–571, 1992.
4. E. R. Carbonnell and D. Kapur. Program verification using automatic generation of invariants. In *Proceedings, International Conference on Theoretical Aspects of Computing '2004*, volume 3407, pages 325–340. Lecture Notes in Computer Science, Springer Verlag, 2004.
5. T. E. Cheatham and J. A. Townley. Symbolic evaluation of programs: A look at loop analysis. In *Proc. of ACM Symposium on Symbolic and Algebraic Computation*, pages 90–96, 1976.
6. M. A. Colon, S. Sankaranarayana, and H. B. Sipna. Linear invariant generation using non linear constraint solving. In *Proceedings, Computer Aided Verification, CAV 2003*, volume 2725 of *Lecture Notes in Computer Science*, pages 420–432. Springer Verlag, 2003.
7. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the Fifth Annual ACM SIGPLAN-SIGACT Symposium on the Principles of Programming Languages*, pages 84–97, 1978.
8. E. Denney and B. Fischer. A generic annotation inference algorithm for the safety certification of automatically generated code. In *Proceedings, the Fifth International Conference on Generative programming and Component Engineering*, Portland, Oregon, 2006.
9. E. Dijkstra. *A Discipline of Programming.* Prentice Hall, 1976.
10. M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 2006.
11. T. Fahringer and B. Scholz. *Advanced Symbolic Analysis for Compilers.* Springer Verlag, Berlin, Germany, 2003.
12. D. Gries. *The Science of programming.* Springer Verlag, 1981.
13. C. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576 – 583, Oct. 1969.
14. M. Karr. Affine relationships among variables of a program. *Acta Informatica*, 6:133–151, 1976.
15. L. Kovacs and T. Jebelean. Automated generation of loop invariants by recurrence solving in theorema. In D. Petcu, V. Negru, D. Zaharie, and T. Jebelean, editors, *Proceedings of the 6th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC04)*, pages 451–464, Timisoara, Romania, 2004. Mirton Publisher.

16. T. J. L. Kovacs. An algorithm for automated generation of invariants for loops with conditionals. In D. P. et. al., editor, *Proceedings of the Computer-Aided Verification on Information Systems Workshop (CAVIS05), 7th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC05)*, pages 16–19, Department of Computer Science, West University of Timisoara, Romania, 2005.
17. Z. Manna. *A Mathematical Theory of Computation*. McGraw Hill, 1974.
18. A. Mili, J. Desharnais, and J. R. Gagne. Strongest invariant functions: Their use in the systematic analysis of while statements. *Acta Informatica*, April 1985.
19. A. Mili, J. Desharnais, and F. Mili. Relational heuristics for the design of deterministic programs. *Acta Informatica*, 24(3):239–276, 1987.
20. A. Mili, M. Pleszkoch, and R. C. Linger. Towards the automated derivation of loop functions. Technical report, New Jersey Institute of Technology, http://web.njit.edu/~ mili/loopx.pdf, 2006.
21. H. Mills. The new math of computer programming. *Communications of the ACM*, 18(1), January 1975.
22. C. Morgan. *Programming from Specifications*. International Series in Computer Sciences. Prentice Hall, London, UK, 1998.
23. J. Morris and B. Wegbreit. Program verification by subgoal induction. In R. Yeh, editor, *Current Trends in Programming Methodology*, volume II, chapter 8. Prentice Hall, Englewood Cliffs, NJ, 1977.
24. S. Sankaranarayana, H. B. Sipna, and Z. Manna. Non linear loop invariant generation using groebner bases. In *Proceedings, ACM SIGPLAN Principles of Programming Languages, POPL 2004*, pages 381–329, 2004.
25. A. Tarski. On the calculus of relations. *The Journal of Symbolic Logic*, 6(3), September 1941.
26. A. Tarski. A lattice-theoretic fixpoint theorem and its applications. *Pacific Journal of Mathematics*, (5):285–309, 1955.