# Invariant Assertions, Invariant Relations, and Invariant Functions

Olfa Mraihi, Asma Louhichi, Lamia Labed Jilani, Jules Desharnais and Ali Mili
Institut Superieur de Gestion, Tunis, Tunisia
Faculte des Sciences de Tunis, El Manar, Tunisia
Laval University, Quebec City, Canada
NJIT, Newark NJ, USA
lamia.labed@isg.rnu.tn, louhichiasma@yahoo.fr, olfa.mraihi@yahoo.fr,
Jules.Desharnais@ift.ulaval.ca, mili@cis.njit.edu

May 1, 2012

**Abstract**

Invariant assertions play an important role in the analysis and documentation of while loops of imperative programs. Invariant functions and invariant relations are alternative analysis tools that are distinct from invariant assertions but are related to them. In this paper we discuss these three concepts and analyze their relationships. The study of invariant functions and invariant relations is interesting not only because it provides alternative means to analyze loops, but also because it gives us insights into the structure of invariant assertions, hence may help us enhance techniques for generating invariant assertions.

**Keywords**

Invariant assertions, invariant functions, invariant relations, loop invariants, program analysis, program verification, while loops, loop functions.

## 1 Introduction

In [31], Hoare introduced *invariant assertions* as a key concept in the analysis of while loops. The study of invariant assertions, and more broadly the functional analysis of programs, have been the focus of active research in the seventies and eighties, and the subject of renewed interest in the last few years [1, 6, 7, 9, 20, 23, 37, 39, 40, 61]. In this paper we wish to investigate the relationships between this well known, thoroughly researched, concept, and two distinct but related concepts: invariant functions [52] and invariant relations [50].

We consider a while loop $w$ on space $S$ defined by $w = \{\texttt{while t do b}\}$. These three concepts can be summarily and informally characterized as follows:

- An invariant assertion is a predicate $\alpha$ on $S$ that is preserved by application of the loop body: if it holds before execution of the loop body, then it holds after.
- An invariant function is a total function on $S$ that takes the same value before and after execution of the loop body (whenever the loop condition holds).

- An invariant relation is a relation that contains all pairs of states $(s, s')$ such that $s'$ is obtained from $s$ by applying an arbitrary number of iterations of the loop body.

The purpose of this paper is three-fold:

- It is a survey of three related but distinct concepts pertaining to the analysis of while loops in C-like programming languages.
- It explores in what sense and to what extent these three concepts offer complementary insights into the functional properties of while loops.
- It offers insights into the structure of invariant assertions, hence may help to streamline the generation of invariant assertions [7, 9, 20, 23, 24, 27, 30, 34, 37, 38, 41, 43, 44, 46, 61, 64].

This paper presents a mix of survey results with original research results; to distinguish these categories, original results are presented with their proofs whereas survey results are presented along with bibliographic references to where their proof can be found.

In section 3, we give definitions of all three concepts, using a uniform model, namely the calculus of relations, which we introduce in section 2. In section 4 we review a number of criteria, and discuss in what way the three classes of invariants are similar, and in what way they are distinct, with respect to each of the selected criteria. In section 5 we discuss the interrelations between these concepts, by asking, for each invariant, whether it can be derived from any of the other invariants, and eventually by what formula. In section 6 we discuss briefly, by means of a sample example, how invariant functions and invariant relations can be used in practice to compute or approximate the function of a while loop, which we take as the semantic definition of the loop. Finally we summarize our results and explore venues of further research, and related work, in section 7.

# 2 Relational Mathematics

The reader is assumed to be familiar with relational mathematics [19, 62]; the main purpose of this section is to introduce notational conventions for standard relational concepts [62].

## 2.1 Elements of Relations

### 2.1.1 Definitions and Notations

We consider a set $S$ defined by the values of some program variables, say $x$, $y$ and $z$; we typically denote elements of $S$ by $s$, and we note that $s$ has the form $s = \langle x, y, z \rangle$. We use the notation $x(s)$, $y(s)$, $z(s)$ to denote the $x$-component, $y$-component and $z$-component of $s$. We may sometimes use $x$ to refer to $x(s)$ and $x'$ to refer to $x(s')$, when this raises no ambiguity. We refer to elements $s$ of $S$ as *program states* and to $S$ as the *state space* (or *space*, for short) of the program that manipulates variables $x$, $y$ and $z$. Given a program $g$ on state space $S$, we use functions on $S$ to capture the function that the program defines from its initial states to its final states, and we use relations on $S$ to capture functional specifications that we may want the program to satisfy. To this effect, we briefly introduce elements of relational mathematics. A relation on $S$ is a subset of the cartesian product $S \times S$. Constant relations on some set $S$ include the *universal* relation, denoted by $L$, the *identity* relation, denoted by $I$, and the *empty* relation, denoted by $\emptyset$.

### 2.1.2 Operations on Relations

Because relations are sets, we apply the usual set theoretic operations between relations: union ($\cup$), intersection ($\cap$), complement ($\overline{R}$), cartesian product ($\times$). Operations on relations also include the *converse*, denoted by $\widehat{R}$, and defined by $\widehat{R} = \{(s, s')|(s', s) \in R\}$. The *product* of relations $R$ and $R'$ is the relation denoted by $R \circ R'$ (or $RR'$) and defined by $R \circ R' = \{(s, s')|\exists s'' : (s, s'') \in R \wedge (s'', s') \in R'\}$. We admit without proof that $\widehat{RR'} = \widehat{R'}\widehat{R}$ and that $\widehat{\widehat{R}} = R$. Given a predicate $t$, we denote by $I(t)$ the subset of the identity relation defined as $I(t) = \{(s, s')|s' = s \wedge t(s)\}$ and by $T$ the relation defined as $T = \{(s, s')|t(s)\}$; by definition, we have $I(t) = I \cap T$ and $T = I(t) \circ L$; adopting the naming convention of [2, 58], we refer to subsets of the identity as *monotypes*. The *pre-restriction* (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 $t$ can be written as $I(t) \circ R$ or $T \cap R$, and the post-restriction of relation $R$ to predicate $t$ can be written as $R \circ I(t)$ or $R \cap \widehat{T}$.

The *domain* of relation $R$ is defined as $dom(R) = \{s|\exists s' : (s, s') \in R\}$. The *range* of relation $R$ is denoted by $rng(R)$ and defined as $dom(\widehat{R})$. We admit without proof that for a relation $R$, $RL = \{(s, s')|s \in dom(R)\}$ and $LR = \{(s, s')|s' \in rng(R)\}$. The *nucleus* of relation $R$ is the relation denoted by $\mu(R)$ and defined as $R\widehat{R}$; this is known in other sources [2, 58] under the name *kernel* (though we reserve the name *kernel* to a related but distinct concept [36]: $\kappa(R) = \{(s, s')|\emptyset \subset s'.R \subseteq s.R\}$). The $n^{th}$ power of relation $R$, for natural number $n$, is denoted by $R^n$ and defined as follows: If $n = 0$ then $I$ else $R \circ R^{n-1}$. We define the *transitive closure* of relation $R$ as the relation denoted by $R^+$ and defined by $R^+ = \{(s, s')|\exists n > 0 : (s, s') \in R^n\}$. We define the *reflexive transitive closure* of relation $R$ as the relation denoted by $R^*$ and defined by $R^* = R^+ \cup I$.

We apply the following conventions with regards to operator precedence: unary operators (complement, inverse, closures) are applied first; they are followed by relational product, then intersection, then union.

### 2.1.3 Relation Taxonomy

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, $RL = L$. A relation $R$ is said to be *reflexive* if and only if $I \subseteq R$, *transitive* if and only if $RR \subseteq R$, *symmetric* if and only if $R = \widehat{R}$ and an *equivalence* if and only if it is reflexive, symmetric, and transitive. We admit without proof that the transitive closure of a relation $R$ is the smallest transitive superset of $R$; and that the reflexive transitive closure of relation $R$ is the smallest reflexive transitive superset of $R$. Given two total deterministic relations $R$ and $R'$, we say that $R$ is *more-injective* than (or as injective as) $R'$ if and only if [52, 57]: $R\widehat{R} \subseteq R'\widehat{R'}$. A total deterministic relation $R$ is said to be *injective* if and only if it is more-injective than $I$. A relation $R$ is said to be *regular* [2, 35, 36] (or difunctional [57, 60]) if and only if $R\widehat{R}R = R$; functions are known to be regular [36, 58].

A relation $R$ is said to be *rectangular* if and only if $R = RLR$. A relation $R$ is said to be a *vector* (also known as a *left condition* [19]) if and only if $RL = R$. In set theoretic terms, a vector on set $S$ has the form $C \times S$, for some subset $C$ of $S$; vector $C \times S$ can also be written as $I(C) \circ L$; we may sometimes use the same symbol to represent a subset $C$ of $S$ and the vector that defines it; it is plain to see that the product of any relation with a vector is a vector. A non-empty vector $p$ on $S$ is said to be a *point* if and only if it satisfies the condition $p\widehat{p} \subseteq I$; whereas a mere vector represents a subset of $C$, a point represents a singleton; the same symbol may be used to represent

a point and the single element of $S$ that defines it. A relation $R$ is said to be *inductive* if and only if it can be written as $R = \overline{A} \cup \widehat{A}$ for some vector $A$; we leave it to the reader to check that if $A$ is written as $\{(s, s')|\alpha(s)\}$, then $\overline{A} \cup \widehat{A}$ can be written as $\{(s, s')|\alpha(s) \Rightarrow \alpha(s')\}$.

### 2.1.4 Relational Laws

In this section, we briefly present some relational laws that will be needed in subsequent discussions. Because these are typically straightforward consequences of definitions, they will be presented without proof. We let $R$ and $Q$ be arbitrary relations, $C$ be an arbitrary vector on $S$, $p$ be a point on $S$, $V$ be a total function on $S$, and $F$ be an arbitrary function on $S$.

(1) $(C \cap Q)R = C \cap QR$,
(2) $\overline{C}L = \overline{C}$,
(3) $L\widehat{C} = \widehat{C}$,
(4) $\widehat{C}R = L(C \cap R)$,
(5) $RC = (R \cap \widehat{C})L$,
(6) $\widehat{\overline{C}}C = \emptyset$,
(7) $(Q \cap \widehat{C})R = Q(C \cap R)$,
(8) $C \cap R = (I \cap C)R = (I \cap \widehat{C})R$,
(9) $\widehat{C} \cap R = R(I \cap \widehat{C}) = R(I \cap C)$,
(10) $I \cap RL = I \cap R\widehat{R} = I \cap L\widehat{R}$,
(11) $Q \subseteq RL \Leftrightarrow QL \subseteq RL$
(12) $Q \subseteq LR \Leftrightarrow LQ \subseteq LR$
(13) $R \subseteq I \Rightarrow \widehat{R} = R$
(14) $R \neq \emptyset \Rightarrow LRL = L$
(15) $\overline{Rp} = \overline{R}p$
(16) $\widehat{pR} = \widehat{p}\overline{R}$
(17) $RV \subseteq Q \Leftrightarrow R \subseteq Q\widehat{V}$
(18) $\widehat{V}R \subseteq Q \Leftrightarrow R \subseteq VQ$
(19) $RF \subseteq Q \Leftrightarrow L\widehat{F} \cap R \subseteq Q\widehat{F}$
(20) $\widehat{F}R \subseteq Q \Leftrightarrow FL \cap R \subseteq FQ$
(21) $Q \cup R(R^*Q) = R^*Q$
(22) $R \subseteq F \wedge FL \subseteq RL \Leftrightarrow R = F$.

We concur with [19, 58] that algebraic manipulations and proofs are superior to pointwise logic reasoning, and will endeavor to cast our results and proofs in the former model, rather than the latter, to the largest extent possible.

### 2.1.5 Relational Invariants

In this section, we introduce two relational concepts, along with a proposition linking them; these concepts are useful for the purpose of our subsequent discussions.

**Definition 1** *We consider a relation $R$ on space $S$ and a subset of $S$ represented by a vector $C$. We say that $R$* preserves vector *$C$ if and only if:*

$$R \cap C \subseteq \widehat{C}.$$

This definition is inspired from the concept of *predicative type assertion*, introduced by Oliveira in [56]. This condition means that if the arguments of $R$ are in (set) $C$, then so are its images.

**Definition 2** *We consider a relation $R$ on space $S$ and a total relation $V$ on $S$. We say that $R$* preserves function *$V$ if and only if:*

$$RV \subseteq V.$$

This definition is inspired from the concept of *invariant function* introduced by Mili et. al. in [52]. This condition means that $V$ takes the same value before and after application of relation $R$. The following Proposition shows how these two concepts are related.

**Proposition 1** *Given a relation $R$ on $S$ and a subset of $S$ represented by vector $C$. If relation $R$ preserves function $V$, then it preserves vector $A = VC$.*

**Proof.** We write the conclusion then show that it is the logical consequence of a tautology.

$$\begin{array}{ll} & VC \cap R \subseteq \widehat{VC} \\ \Leftrightarrow & \quad \{ \text{ converse of a product } \} \\ & VC \cap R \subseteq \widehat{C}\widehat{V} \\ \Leftrightarrow & \quad \{ \text{ identity 17 } \} \\ & (VC \cap R)V \subseteq \widehat{C} \\ \Leftrightarrow & \quad \{ \text{ identity 1, } VC \text{ is a vector } \} \\ & VC \cap RV \subseteq \widehat{C} \\ \Leftarrow & \quad \{ \ R \text{ preserves function } V \ \} \\ & VC \cap V \subseteq \widehat{C} \\ \Leftrightarrow & \quad \{ \text{ rewriting the left-hand side as a product } \} \\ & (I \cap VC)V \subseteq \widehat{C} \\ \Leftrightarrow & \quad \{ \text{ identity 17 } \} \\ & (I \cap VC) \subseteq \widehat{C}\widehat{V} \\ \Leftrightarrow & \quad \{ \text{ taking the converse on both sides } \} \\ & (I \cap VC) \subseteq VC \\ \Leftrightarrow & \quad \{ \text{ set theory } \} \\ & \text{true.} \end{array}$$

**qed**

To illustrate the meaning of this proposition, we take a simple example of set $S$ defined by three integer variables $x$, $y$, and $z$, and we let $R$, $V$ and $C$ be defined as follows:

$$
\begin{aligned}
&R = \{(s,s')|x' = x+1 \wedge y' = y-1\},\\
&V = \{(s,s')|x' = x+y \wedge y' = 0 \wedge z' = 0\},\\
&C = \{(s,s')|x+y = 10\}.
\end{aligned}
$$

It is plain that $V$ is total and deterministic and that $C$ is a vector. To check that $R$ preserves function $V$, we compute $RV$:

$$
\begin{aligned}
& RV\\
= & \quad \{ \text{ substitutions } \}\\
& \{(s,s')|\exists s\text{"} : x\text{"} = x+1 \wedge y\text{"} = y-1 \wedge x' = x\text{"} + y\text{"} \wedge y' = 0 \wedge z' = 0\}\\
= & \quad \{ \text{ substitution, simplification } \}\\
& \{(s,s')|x' = x+y \wedge y' = 0 \wedge z' = 0\}\\
= & \quad \{ \text{ substitution } \}\\
& V.
\end{aligned}
$$

According to this proposition, relation $R$ preserves $A = VC$. We write $A$ as:

$$
\begin{aligned}
& VC\\
= & \quad \{ \text{ substitution } \}\\
& \{(s,s')|\exists s\text{"} : x\text{"} = x+y \wedge y\text{"} = 0 \wedge z\text{"} = 0 \wedge x\text{"} = 10\}\\
= & \quad \{ \text{ substitution, simplification } \}\\
& \{(s,s')|x+y = 10\}.
\end{aligned}
$$

We briefly check that $R$ preserves $A$ (though this is assured by Proposition 1):

$$
\begin{aligned}
& A \cap R\\
= & \quad \{ \text{ substitutions } \}\\
& \{(s,s')|x+y = 10 \wedge x' = x+1 \wedge y' = y-1\}\\
= & \quad \{ \text{ equivalence } \}\\
& \{(s,s')|x+y = 10 \wedge x' = x+1 \wedge y' = y-1 \wedge x'+y' = 10\}\\
\subseteq & \quad \{ \text{ logic } \}\\
& \{(s,s')|x'+y' = 10\}\\
= & \quad \{ \text{ substitution } \}\\
& \widehat{A}.
\end{aligned}
$$

### 2.2 Refinement Ordering

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

**Definition 3** *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 each element $s$ in the domain of $R'$, the set of images of $s$ by $R$ is a subset of (or equal to) the set of images of $s$ by $R'$. This is similar to, but different from, refining a pre/postcondition specification by weakening its precondition and/or strengthening its postcondition [18, 25, 47, 54]. We denote this relation by $R \sqsupseteq R'$ or $R' \sqsubseteq R$. Intuitively, $R$ refines $R'$ if and only if $R$ represents a stronger requirement than $R'$.

### 2.3 Lattice Properties

We admit without proof that the refinement relation is a partial ordering. In [4] 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:

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

  The same result is found, constructively, by Oliveira and Rodrigues [58], using point-free relational algebra.

- Two relations $R$ and $R'$ have a least upper bound if and only if they satisfy the following condition (which we refer to as the *consistency 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:

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

  Intuitively, the join of $R$ and $R'$, when it exists, behaves like $R$ outside the domain of $R'$, behaves like $R'$ outside the domain of $R$, and behaves like the intersection of $R$ and $R'$ on the intersection of their domain. The consistency condition ensures that the domain of their intersection is identical to the intersection of their domains; this condition is computed constructively by Oliveira and Rodrigues [58] using point-free relational algebra.

- 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.

This lattice structure was further analyzed by Oliveira and Rodrigues [58], using a constructive approach based on point-free relational algebra (by contrast with the analysis of Boudriga et. al. [4], which was mostly pointwise).

# 3 Formal Definitions

In this section, we present formal definitions of invariant assertions, invariant relations and invariant functions; prior to these definitions, we introduce, in section 3.1, some notations and background results about loops. For the sake of uniformity, we use the same mathematical baggage to represent all three types of invariants, namely the calculus of relations. Also, for the sake of illustration, we use a running example in the form of a simple loop on three natural variables $n$, $f$, $k$, such that $1 \leq k \leq n+1$:

$$w: \ \textbf{while}\ (k \neq n+1)\ \{f = f * k; k = k+1;\}.$$

## 3.1 Loop Semantics

We consider a deterministic program $g$ on some variables $x_1$, $x_2$, ..., $x_n$ in some C-like programming language. We let $S$ be the space defined by all the values that the aggregate of variables may take. Because $g$ is deterministic, its semantics is captured by a function, which we denote by $[g]$ and define by

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

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

$$dom([g]) = \{s|\ \text{if } g \text{ starts execution in state } s \text{ then it terminates}\}.$$

As a general convention, we represent programs by lower case letters (e.g. $g$), and represent their function by the same letter, in upper case (e.g. $G$).

We focus specifically on iterative programs written in some C-like programming language, of the form $w = \{\texttt{while t do b}\}$, and we present some results about such programs, which we use subsequently. Without (much) loss of generality, we assume that the programming languages under consideration are deterministic, or at least we limit our attention to deterministic constructs of these languages. Furthermore, we restrict our discussion to while loops that terminate for all states in their space. We briefly discuss our argument, presented in [51], to the effect that this hypothesis causes no loss of generality. Given a while loop $w$ on space $S$, we let $s$, be an arbitrary element of $dom(W)$. By definition, execution of $w$ on state $s$ terminates; let $s_i$, for $i \geq 1$ be the state obtained after $i$ iterations of the loop starting at $s$, and let $s_F$ be the final state in the execution of $w$ on $s$. Execution of $w$ on $s_F$ terminates (immediately, in fact, since $t(s_F)$ is false), and execution of the loop on any of the intermediate states $s_i$ also terminates, since if it did not, neither would execution of $w$ on $s$. Hence all the intermediate states and the final state of the execution of $w$ on $s$ are in $dom(W)$. Therefore by restricting $S$ to $dom(W)$ we are not excluding any state of interest (all initial states, intermediate states and final states of the loop are in $dom(W)$).

Given a while loop $w$ on space $S$, we redefine the space as $S := dom(W)$, thereby making $W$ vacuously total on (the newly defined) space $S$. Given the space $S$ of the while loop, we can compute the domain of $W$ (as a subset of $S$) using any of the methods and tools available today, such as those presented in [5, 10–12, 28, 29, 45, 63].

**Theorem 1 Mills Theorem** *(H.D. Mills [53]). We consider a while loop* $w = \{\texttt{while t do b}\}$ *on space* $S$ *that terminates for all states in* $S$*, and a function* $W$ *on the same space* $S$*, then* $[w] = W$ *if and only if:*

- $WL = L$.
- $\overline{T} \cap W = \overline{T} \cap I$.
- $T \cap W = T \cap BW$.

We rewrite the second and third conditions of this theorem using monotypes, as follows:

- $I(\neg t) \circ W = I(\neg t)$.
- $I(t) \circ B \circ W = I(t) \circ W$.

The following theorem gives an explicit expression of the loop function and represents, for our purposes, the semantic definition of while loops. This theorem is a verbatim representation, in relational terms, of the loop's operational semantics [3, 48, 49].

**Theorem 2** *Given a while statement of the form* $w = \{\texttt{while t do b}\}$ *that terminates for all the states in* $S$*. Then its function* $W$ *is given by:*

$$W = (T \cap B)^* \cap \widehat{\overline{T}}.$$

**Proof.** We denote $(T \cap B)^* \cap \widehat{\overline{T}}$ by $W'$ and resolve to prove that $W'$ so defined equals $W$, the function of the loop. Given that $W$ is assumed to be a function (since we consider loops written in a deterministic programming language), it suffices (according to law 22) to prove two conditions:

- $W' \subseteq W$,
- $WL \subseteq W'L$.

Since the while loop is supposed to terminate for all states in $S$, we write:

$$\begin{array}{ll} & \forall s \in S : \exists s' \in S : \exists n \geq 0 : (s, s') \in (T \cap B)^n \wedge \neg t(s') \\ \Leftrightarrow & \quad \{ \text{ definition of transitive closure } \} \\ & \forall s \in S : \exists s' \in S : (s, s') \in (T \cap B)^* \wedge \neg t(s') \\ \Leftrightarrow & \quad \{ \text{ recasting in relational terms } \} \\ & \forall (s, s') \in L : (s, s') \in (T \cap B)^* \circ \overline{T} \\ \Leftrightarrow & \quad \{ \text{ set theory } \} \\ & L \subseteq (T \cap B)^* \circ \overline{T} \\ \Leftrightarrow & \quad \{ \ W \text{ is total, vector identity 5 } \} \\ & WL \subseteq W'L. \end{array}$$

This proves the second condition, as for the first condition, we begin by rewriting $W'$ as follows, using monotypes rather than vectors:

$$W' = (I(t) \circ B)^* \circ I(\neg t)$$

and we proceed as follows:

$W = IW$

$\Leftrightarrow$ { decomposing the identity, distributivity }

$W = I(\neg t) \circ W \cup I(t) \circ W$

$\Leftrightarrow$ { Mills' Theorem, associativity }

$W = I(\neg t) \cup (I(t) \circ B) \circ W$

$\Rightarrow$ { Kleene algebra: least fixpoint }

$(I(t) \circ B)^* \circ I(\neg t) \subseteq W$

$\Leftrightarrow$ { substitution }

$W' \subseteq W$.

**qed**

### 3.2 Invariant Assertions

Traditionally [22, 31, 47], an invariant assertion $\alpha$ for the while loop

$$w = \{ \textbf{while } t \textbf{ do } b\}$$

with respect to a precondition/ postcondition pair ($\phi$, $\psi$) is defined as a predicate on $S$ that satisfies the following conditions:

- $\phi \Rightarrow \alpha$.
- $\{\alpha \wedge t\} b \{\alpha\}$.
- $\alpha \wedge \neg t \Rightarrow \psi$.

As defined, the invariant assertion is dependent not only on the while loop, but also on the loop's specification, in the form of a precondition/ postcondition pair. This precludes meaningful comparisons with invariant relations and invariant functions, which are dependent solely on the loop. Hence we redefine the concept of invariant assertion in terms of the second condition alone. Also, to represent an invariant assertion, we map the predicate on $S$ into a vector (a relation) on $S$. Specifically, we represent the predicate $\alpha$ by the vector $A$ defined by

$$A = \{(s, s') | \alpha(s)\}.$$

**Definition 4** *Given a while statement on space $S$ of the form*

$$w = \{\texttt{while t do b}\}$$

*that terminates for all initial states in $S$, and a vector $A$ on $S$, we say that $A$ is an* invariant assertion *for $w$ if and only if $(T \cap B)$ preserves vector $A$.*

In other words, $A$ is an invariant assertion for $w$ if and only if:

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

This is a straightforward interpretation, in relational terms, of the second condition of Hoare's rule,

$$\{\alpha \wedge t\}B\{\alpha\}.$$

For the running example, we claim that the following vector satisfies the condition of Definition 4:

$$A = \{(s, s')|f = (k-1)!\}.$$

To verify that $A$ is an invariant assertion, we compute the left hand side of the definition:

$$\begin{array}{ll} & A \cap T \cap B \\ = & \quad \{ \text{ substitutions } \} \\ & \{(s,s')|f = (k-1)! \wedge k \neq n+1 \wedge n' = n \wedge f' = f \times k \wedge k' = k+1\} \\ \subseteq & \quad \{ \text{ deleting conjuncts } \} \\ & \{(s,s')|f = (k-1)! \wedge f' = f \times k \wedge k' = k+1\} \\ = & \quad \{ \text{ substitution } \} \\ & \{(s,s')|f = (k-1)! \wedge f' = (k'-1)! \wedge k' = k+1\} \\ \subseteq & \quad \{ \text{ deleting conjuncts } \} \\ & \{(s,s')|f' = (k'-1)!\} \\ = & \quad \{ \text{ substitution } \} \\ & \widehat{A}. \end{array}$$

Note that we have not proved that the assertion $f = (k-1)!$ holds after each iteration; rather we have only proved that if this assertion holds at one iteration, then it holds at the next iteration, hence (by induction) after each iteration thereafter. This is in effect an inductive proof without a basis of induction.

### 3.3 Invariant Functions

Invariant functions are functions whose value remains unchanged by application of the loop body's function [52].

**Definition 5** *Let $w$ be a while statement of the form* {`while t do b`} *that terminates normally for all initial states in $S$, and let $V$ be a total function on $S$. We say that $V$ is an* invariant function *for $w$ if and only if $(T \cap B)$ preserves function $V$.*

In other words, an invariant function is a total function $V$ on $S$ that satisfies the condition $(T \cap B)V \subseteq V$. The following Proposition provides an alternative characterization.

**Proposition 2** *Let $w$ be a while statement of the form* {`while t do b`} *that terminates normally for all initial states in $S$, and let $V$ be a total function on $S$. Function $V$ on $S$ is an* invariant function *of $w$ if and only if:*

$$T \cap BV = T \cap V.$$

**Proof.** *Sufficiency.* From $T \cap BV = T \cap V$ we infer (by identity 1) $(T \cap B)V = T \cap V$ from which we infer (by set theory) $(T \cap B)V \subseteq V$.

*Necessity.* Since $T \cap V$ is a function, we can prove $T \cap BV = T \cap V$ by proving $T \cap BV \subseteq T \cap V$ and $(T \cap V)L \subseteq (T \cap BV)L$. We proceed as follows: From $(T \cap B)V \subseteq V$ we infer (by identity 1) $T \cap BV \subseteq V$. Combining this with the set theoretic identity $T \cap BV \subseteq T$, we find (by set theory) $T \cap BV \subseteq T \cap V$.

As for proving that $(T \cap V)L$ is a subset of $(T \cap BV)L$, we note that because $w$ terminates for all states in $S$, $T$ is necessarily a subset of (or equal to) $BL$ (if not any state in $T \setminus BL$ will cause the loop not to terminate), and we proceed as follows:

$$
\begin{array}{ll}
 & (T \cap V)L \subseteq (T \cap BV)L \\
\Leftrightarrow & \quad \{ \text{ identity 1, applied twice } \} \\
 & T \cap VL \subseteq T \cap BVL \\
\Leftrightarrow & \quad \{ \text{ totality of } V, \text{ applied twice } \} \\
 & T \cap L \subseteq T \cap BL \\
\Leftrightarrow & \quad \{ \text{ left: algebra; right: hypothesis } T \subseteq BL \} \\
 & T \subseteq T \\
\Leftrightarrow & \quad \{ \text{ set theory } \} \\
 & \text{true.}
\end{array}
$$

**qed**

This condition can also be written using monotypes rather than vectors:

$$I(t) \circ V = I(t) \circ B \circ V.$$

To illustrate the concept of invariant function, we consider the loop of the running example, and propose the following function:

$$V \begin{pmatrix} n \\ f \\ k \end{pmatrix} = \begin{pmatrix} \frac{f}{(k-1)!} \\ 0 \\ 0 \end{pmatrix}.$$

This function is total, since its value can be computed for any state in $S$. To check the preservation condition, we compute:

$$
\begin{array}{ll}
 & T \cap (BV) \\
\subseteq & \quad \{ \text{ substitution, simplification } \} \\
 & \{(s,s') | n' = n \wedge f' = f \times k \wedge k' = k+1\} \circ \{(s,s') | n' = \frac{f}{(k-1)!} \wedge f' = 0 \wedge k' = 0\} \\
= & \quad \{ \text{ relational product } \} \\
 & \{(s,s') | n' = \frac{f \times k}{(k+1-1)!} \wedge f' = 0 \wedge k' = 0\} \\
= & \quad \{ \text{ simplification } \} \\
 & \{(s,s') | n' = \frac{f}{(k-1)!} \wedge f' = 0 \wedge k' = 0\} \\
= & \quad \{ \text{ substitution } \} \\
 & V.
\end{array}
$$

### 3.4 Invariant Relations

Intuitively, an invariant relation is a relation that contains all the pairs of states $(s, s')$ such that $s'$ can be derived from $s$ by application of an arbitrary number (including zero) of iterations of the loop body. We define it formally as follows.

**Definition 6** *Given a while loop of the form* $w = \{\texttt{while t do b}\}$ *on some space* $S$*, and given a relation* $R$ *on* $S$*, we say that* $R$ *is an* invariant relation *for* $w$ *if and only if it is a reflexive and transitive superset of* $(T \cap B)$*.*

As a rationale for the concept of invariant relation, consider that a reflexive transitive closure of a relation is the smallest reflexive transitive superset of the relation. Had we been able to derive the reflexive transitive closure of $(T \cap B)$, we would apply Theorem 2 to compute the function of the loop. But computing the reflexive transitive closure of $(T \cap B)$, i.e. the *smallest* reflexive transitive superset of $(T \cap B)$, is usually very difficult; as a substitute, invariant relations provide arbitrary (not necessarily smallest) reflexive transitive supersets of $(T \cap B)$. By taking the intersection of a sufficient number of invariant relations, we may attain or approximate the reflexive transitive closure of $(T \cap B)$, from which we can compute or approximate the function of the loop.

To illustrate this concept, we consider again the loop of the running example, and we propose the following relation:

$$R = \left\{ (s, s') | \frac{f}{(k-1)!} = \frac{f'}{(k'-1)} \right\}.$$

This relation is reflexive and transitive, since it is the nucleus of a function. To check the invariance condition, we compute the intersection $T \cap B \cap R$ and check that it equals $T \cap B$.

$$\begin{array}{ll} & T \cap B \cap R \\ = & \quad \{ \text{ Substitution } \} \\ & \{(s, s') | k \neq n+1 \wedge n' = n \wedge f' = f \times k \wedge k' = k+1 \wedge \frac{f}{(k-1)!} = \frac{f'}{(k'-1)!}\} \\ = & \quad \{ \text{ simplification } \} \\ & \{(s, s') | k \neq n+1 \wedge n' = n \wedge f' = f \times k \wedge k' = k+1\} \\ = & \quad \{ \text{ substitution } \} \\ & T \cap B. \end{array}$$

## 4 Comparative Analysis

In this section, we go beyond the definitions and attempt to characterize invariant assertions, invariant relations and invariant functions in such a way as to highlight their similarities and differences. We consider in turn a number of criteria, and see how each type of invariant fares with respect to each criterion.

### 4.1 Invariants as Supersets

According to Definition 6, an invariant relation is a reflexive transitive superset of $(T \cap B)$. In this section, we show that, interestingly, the search for invariant assertions and the search for invariant functions can both be seen as searching for a reflexive transitive superset of $(T \cap B)$.

### 4.1.1 Invariant Assertions as Supersets

We put forth the following proposition.

**Proposition 3** *Let $A$ be an invariant assertion for while statement $w = \{\texttt{while t do b}\}$ on space $S$. Then $(\overline{A} \cup \widehat{A})$ is a reflexive transitive superset of $(T \cap B)$.*

**Proof.** Reflexivity of $(\overline{A} \cup \widehat{A})$ can be established as follows:

$$
\begin{array}{ll}
& I \subseteq \overline{A} \cup \widehat{A} \\
\Leftrightarrow & \quad \{ \text{ set theory } \} \\
& I \cap A \subseteq \widehat{A} \\
\Leftrightarrow & \quad \{ \text{ taking the converse on both sides } \} \\
& \widehat{I \cap A} \subseteq A \\
\Leftrightarrow & \quad \{ \text{ Identity 13 } \} \\
& I \cap A \subseteq A.
\end{array}
$$

Because the last statement in this chain of equivalences is valid (by set theory), so is the first.

Transitivity of $(\overline{A} \cup \widehat{A})$ can be established as follows:

$$
\begin{array}{ll}
& (\overline{A} \cup \widehat{A})(\overline{A} \cup \widehat{A}) \\
\subseteq & \quad \{ \text{ distributivity and monotonicity } \} \\
& \overline{A} L \cup \widehat{A}(\overline{A} \cup \widehat{A}) \\
= & \quad \{ \text{ Identities 2 and 4 } \} \\
& \overline{A} \cup L(A \cap (\overline{A} \cup \widehat{A})) \\
= & \quad \{ \text{ Distributivity and Boolean algebra } \} \\
& \overline{A} \cup L(A \cap \widehat{A}) \\
\subseteq & \quad \{ \text{ monotonicity and Identity 3 } \} \\
& \overline{A} \cup \widehat{A}.
\end{array}
$$

To prove that $(\overline{A} \cup \widehat{A})$ is a superset of $(T \cap B)$, we merely refer to the definition:

$$
\begin{array}{ll}
& A \cap T \cap B \subseteq \widehat{A} \\
\Leftrightarrow & \quad \{ \text{ set theory } \} \\
& T \cap B \subseteq (\overline{A} \cup \widehat{A}).
\end{array}
$$

**qed**

### 4.1.2 Invariant Functions as Supersets

We put forth the following proposition.

**Proposition 4** *Let $V$ be an invariant function for while statement $w = \{\texttt{while t do b}\}$ on space $S$. Then $(V\widehat{V})$ is a reflexive transitive superset of $(T \cap B)$.*

**Proof.** Because $V$ is total, its nucleus is reflexive; because $V$ is deterministic, its nucleus is transitive. Because $(T \cap B)$ preserves function $V$, we know that $(T \cap B)V \subseteq V$, from which we infer (by identity 17) $(T \cap B) \subseteq V\widehat{V}$. **qed**

Note that if $V$ is a total function (which invariant functions are), then the nucleus of $V$ ($V\widehat{V}$) has the form $\{(s, s')|V(s) = V(s')\}$.

#### 4.1.3 Taking Stock

We want to reflect on the results of Propositions 3 and 4. These Propositions provide that the search for invariant assertions and the search for invariant functions are both specialized forms of the search for invariant relations. But while the search for invariant assertions seeks an inductive relation (of the form $(\overline{A} \cup \widehat{A})$ for some vector $A$) and the search for invariant functions seeks an equivalence relation (of the form $V\widehat{V}$, for some function $V$), the search for invariant relations seeks an arbitrary reflexive transitive relation that is a superset of $(T \cap B)$. This suggests that invariant relations are a more general concept than invariant assertions and invariant functions —a claim which will be further discussed in section 5.

It is possible that reflexivity and transitivity are the only two properties that inductive relations (stemming from invariant assertions) and equivalence relations (stemming from invariant functions) have in common. This would make invariant relations some kind of greatest lower bound of invariant assertions and invariant functions in the IsA lattice.

### 4.2 Invariants and the Loop Function

Because we represent the semantics of a loop by the function that the loop computes, it is worthwhile to explore the relationships between the three invariants of interest and the loop function. This is the subject of this section.

#### 4.2.1 Invariant Assertions and Loop Functions

The following proposition elucidates a simple relation between the function of the loop and a possible invariant assertion of the loop.

**Proposition 5** *Let $w$ be the while loop* {`while t do b`} *on space $S$ and let $C$ be a subset of $S$. Then the vector $A$ on $S$ defined by $A = WC$ is an invariant assertion for $w$, where $W$ is the function of $w$.*

**Proof.** According to Mills' Threorem, the loop function $W$ satifies the condition $T \cap W = T \cap BW$. According to Proposition 2, this is equivalent to $T \cap BW \subseteq W$, which we rewrite (according to identity 1) as $(T \cap B)W \subseteq W$. According to Proposition 1, we infer from this that relation $(T \cap B)$ preserves vector $WC$, which by Definition 4 means that $WC$ is an invariant assertion for $w$. **qed**

Note that, interestingly, this proposition holds for any subset $C$ of $S$; we need to remember the inductive nature of our definition of invariant assertion (Definition 4). We are not saying that $A$ holds necessarily after each iteration; rather we are only saying that if it holds at some iteration, it holds subsequently after any number of iterations. In section 4.4, we discuss how to choose $C$ in such a way that the invariant assertion holds initially (hence after each iteration thereafter).

### 4.2.2 Invariant Relations and Loop Function

In this section, we put forth two propositions that elucidate the relation between invariant relations and loop functions. The first proposition shows us how to derive an invariant relation from the function of the loop.

**Proposition 6** *Given a while loop $w$ on space $S$ of the form* {`while t do b`}*; we assume that $w$ terminates for all $s$ in $S$, and we let $W$ be the function defined by $w$. Then $R = \mu(W)$ is an invariant relation for $w$.*

**Proof.** Relation $R = \mu(W)$ is reflexive and transitive because $W$ is total and deterministic. According to Mills' Theorem, the loop function $W$ satifies the condition $T \cap W = T \cap BW$. According to Proposition 2, this is equivalent to $T \cap BW \subseteq W$, which we rewrite (according to identity 1) as $(T \cap B)W \subseteq W$. By identity 17, this is equivalent to $(T \cap B) \subseteq W\widehat{W}$. **qed**

Proposition 6 shows us how to derive an invariant relation from the loop function; a much more useful result in practice is how to derive the function of the loop (or an approximation thereof) from an invariant relation. This is the subject of the following proposition.

**Proposition 7** *We consider a while loop $w$ on space $S$ of the form* {`w: while t do b`}*, which terminates for any element in $S$. If $R$ is an invariant relation of $w$ then $W$ refines $R \cap \widehat{\overline{T}}$.*

**Proof.** First, we introduce a simple lemma. By hypothesis, $R$ is a superset of $(T \cap B)$. We write:

$$
\begin{array}{ll}
 & (T \cap B) \subseteq R \\
\Rightarrow & \quad \{ \text{ taking the closure on both sides } \} \\
 & (T \cap B)^* \subseteq R^* \\
\Rightarrow & \quad \{ \text{ right hand side is reflexive and transitive } \} \\
 & (T \cap B)^* \subseteq R \\
\Rightarrow & \quad \{ \text{ monotonicity } \} \\
 & (T \cap B)^* \cap \widehat{\overline{T}} \subseteq R \cap \widehat{\overline{T}} \\
\Rightarrow & \quad \{ \text{ Theorem 2 } \} \\
 & W \subseteq R \cap \widehat{\overline{T}}.
\end{array}
$$

Now, we let $Y$ be $R \cap \widehat{\overline{T}}$. In order to prove that $W \sqsupseteq Y$, we use Definition 3:

$$
\begin{array}{ll}
 & WL \cap YL \cap (W \cup Y) \\
= & \quad \{ \text{ lemma above: } W \subseteq Y \} \\
 & WL \cap YL \cap (Y) \\
= & \quad \{ \text{ identity: } Y \subseteq YL \} \\
 & WL \cap Y \\
= & \quad \{ \text{ hypothesis: } WL = L \} \\
 & Y.
\end{array}
$$

Hence $W \sqsupseteq Y$. **qed**

The interest of this proposition is that it enables us to use any invariant relation to build a lower bound for the function of the loop. Because the function of the loop is total and deterministic, it is maximal in the lattice of refinement. Hence we can compute or approximate it using only lower bounds. This will be further illustrated in section 6.

Hence, to summarize, we can derive an invariant relation from the loop function, and we can approximate (provide a lower bound of) the loop function from an invariant relation.

#### 4.2.3 Invariant Functions and Loop Function

The relation between invariant functions and loop functions is summarized in the following proposition.

**Proposition 8** *Given a while loop $w$ on $S$ that terminates for all $s$ in $S$,*

- *The function of the loop ($W$) is an invariant function for $w$.*
- *The function of the loop ($W$) is* more-injective *than any invariant function.*

**Proof.** The first premise stems readily from the definition of invariant functions and from Mills' Theorem. To prove the second premise, we let $V$ be an invariant function and we resolve to prove that $W$ (the function of the loop) is more-injective than $V$, i.e. that $W\widehat{W} \subseteq V\widehat{V}$. We proceed as follows, starting from the result of Proposition 4.

$$
\begin{array}{ll}
 & (T \cap B) \subseteq V\widehat{V} \\
\Rightarrow & \quad \{ \text{ taking the reflexive transitive closure on both sides } \} \\
 & (T \cap B)^* \subseteq (V\widehat{V})^* \\
\Rightarrow & \quad \{ \text{ because } (V\widehat{V}) \text{ is reflexive and transitive } \} \\
 & (T \cap B)^* \subseteq (V\widehat{V}) \\
\Rightarrow & \quad \{ \text{ taking the nucleus on both sides } \} \\
 & (T \cap B)^*\widehat{(T \cap B)^*} \subseteq (V\widehat{V})\widehat{(V\widehat{V})} \\
\Rightarrow & \quad \{ \text{ associativity, simplification } \} \\
 & (T \cap B)^*\widehat{(T \cap B)^*} \subseteq (V\widehat{V}V)\widehat{V} \\
\Rightarrow & \quad \{ \text{ because } V \text{ is difunctional } \} \\
 & (T \cap B)^*\widehat{(T \cap B)^*} \subseteq V\widehat{V} \\
\Rightarrow & \quad \{ \text{ rewriting } \} \\
 & (T \cap B)^* \circ I \circ \widehat{(T \cap B)^*} \subseteq V\widehat{V} \\
\Rightarrow & \quad \{ \text{ because } I \supseteq I(\neg t), \text{ and } I(\neg t) = I(\neg t) \circ I(\neg t), \text{ and } I(\neg t) = \widehat{I(\neg t)} \ \} \\
 & (T \cap B)^* \circ I(\neg t) \circ \widehat{I(\neg t)} \circ \widehat{(T \cap B)^*} \subseteq V\widehat{V} \\
\Rightarrow & \quad \{ \text{ rewriting } I(\neg t) \text{ as } I \cap \overline{T}, \text{ and using the identity } \widehat{AB} = \widehat{B}\widehat{A} \ \} \\
 & ((T \cap B)^* \circ (I \cap \overline{T})) \circ \widehat{((T \cap B)^* \circ (I \cap \overline{T}))} \subseteq V\widehat{V} \\
\Rightarrow & \quad \{ \text{ identity 9 } \}
\end{array}
$$

$$
\begin{array}{ll}
 & ((T \cap B)^* \cap \widehat{\overline{T}}) \circ (\widehat{(T \cap B)^* \cap \overline{T}}) \subseteq V\widehat{V} \\
\Rightarrow & \quad \{ \text{ Theorem 2 } \} \\
 & W\widehat{W} \subseteq V\widehat{V}
\end{array}
$$

**qed**

In summary, the function of a loop is an invariant function for the loop, and a most injective invariant function (*strongest* invariant function [52]).

## 4.3 Ordering Invariants

Invariants are not created equal. For example **true** is an invariant assertion for any loop (according to the context-free criterion of Definition 4), though a totally useless/ uninformative assertion; the full relation ($L$) is an invariant relation for any loop, though a totally useless/ uninformative invariant relation; finally a constant function is an invariant function for any loop, though a totally useless/ uninformative invariant function. In this section we briefly review how each type of invariant is ordered.

### 4.3.1 Ordering Invariant Assertions by Implication

Invariant assertions are naturally ordered by logical implication. As we recall from Definition 4, the invariant assertions are represented by vectors; hence stronger invariant assertions are represented by smaller vectors. The inclusion ordering among vectors defines a lattice structure, where the join is the intersection of vectors and the meet is the union of vectors. For the sake of illustration, we consider the following invariant assertions, say $A_0$ and $A_1$, defined as:

$$A_0 = \{(s, s')|f = (k-1)!\}$$

$$A_1 = \{(s, s')|f = (k-1)! \wedge n = 20\}.$$

To check that $A_1$ is an invariant assertion of the loop, we compute the expression $A_1 \cap T \cap B$. We find,

$$
\begin{array}{ll}
 & A_1 \cap T \cap B \\
= & \quad \{ \text{ substitutions } \} \\
 & \{(s, s')|f = (k-1)! \wedge n = 20 \wedge k \neq n+1 \wedge n' = n \wedge f' = f \times k \wedge k' = k+1\} \\
= & \quad \{ \text{ substitutions } \} \\
 & \{(s, s')|f = (k-1)! \wedge n = 20 \wedge k \neq n+1 \wedge n' = 20 \wedge f' = (k-1)! \times k \wedge k' = k+1\} \\
= & \quad \{ \text{ substitutions, simplifications } \} \\
 & \{(s, s')|f = (k-1)! \wedge n = 20 \wedge k \neq n+1 \wedge n' = 20 \wedge f' = (k'-1)! \wedge k' = k+1\} \\
\subseteq & \quad \{ \text{ set theory } \} \\
 & \{(s, s')|n' = 20 \wedge f' = (k'-1)!\} \\
= & \quad \{ \text{ substitution } \} \\
 & \widehat{A_1}.
\end{array}
$$

We leave it to the reader to check that $A_0$ is also an invariant assertion. Also, because $A_1$ is a subset of $A_0$, we concur that $A_1$ represents a stronger invariant assertion of the loop.

### 4.3.2 Ordering Invariant Functions by Injectivity

Invariant functions are total by definition, hence they all have the same domain. When we write an equation such as

$$V(s) = V(B(s)),$$

this gives all the more information on $B$ that $V$ is more injective, i.e. partitions its domain finely. If $V$ were a constant, then this equation does not tell us anything about $B$, since it holds for all $B$. Hence we order invariant functions by injectivity.

To illustrate this ordering, we consider the following two invariant functions, and we leave it to the reader to check that they are indeed invariant functions for the running sample program:

$$V_0 \begin{pmatrix} n \\ f \\ k \end{pmatrix} = \begin{pmatrix} 0 \\ \frac{f}{(k-1)!} \\ 0 \end{pmatrix}.$$

$$V_1 \begin{pmatrix} n \\ f \\ k \end{pmatrix} = \begin{pmatrix} n \\ \frac{f}{(k-1)!} \\ 0 \end{pmatrix}.$$

To check which (if any) of $V_0$ and $V_1$ is more-injective than the other, we compute their nuclei, and find

$$\mu(V_0) = \left\{ (s, s') | \frac{f}{(k-1)!} = \frac{f'}{(k'-1)!} \right\}.$$

$$\mu(V_1) = \left\{ (s, s') | \frac{f}{(k-1)!} = \frac{f'}{(k'-1)!} \wedge n' = n \right\}.$$

We infer that $V_1$ is more-injective than $V_0$, since $\mu(V_1) \subseteq \mu(V_0)$. Indeed, $V_1$ represents more invariant information than $V_0$.

### 4.3.3 Ordering Invariant Relations by Refinement

Invariant relations are ordered by refinement, which, as we have discussed in section 2.3, has lattice-like properties. More refined relations give more information on loop behavior. Because they are by definition reflexive, invariant relations are total. If we consider the definition of refinement (Definition 3), we find that it can be simplified as follows

$$\begin{aligned}
& RL \cap R'L \cap (R \cup R') = R' \\
\Leftrightarrow & \qquad \{ \ R \text{ and } R' \text{ are total } \} \\
& L \cap L \cap (R \cup R') = R' \\
\Leftrightarrow & \qquad \{ \text{ Set Theory } \} \\
& R \cup R' = R' \\
\Leftrightarrow & \qquad \{ \text{ Set Theory } \}
\end{aligned}$$

$R \subseteq R'$.

Hence for invariant relations, refinement is synonymous with set inclusion. We illustrate this ordering with a simple example. We leave it to the reader to check that the following two relations are invariant relations for our running sample program.

$$R_0 = \{(s, s') | \frac{f}{(k-1)!} = \frac{f'}{(k'-1)!}\}.$$

$$R_1 = \{(s, s') | \frac{f}{(k-1)!} = \frac{f'}{(k'-1)!} \wedge n = n'\}.$$

Invariant relation $R_1$ is a subset of, therefore (because they are both total) a refinement of, invariant relation $R_0$. Clearly, the latter also provides more information on loop properties than the former.

## 4.4 Strongest Invariants

In this section we briefly review the general form of strongest invariants for a given loop. As far as invariant functions are concerned, we already know that the loop function is an invariant function, and that it is a most injective invariant function; hence we will only discuss invariant assertions and invariant relations.

### 4.4.1 Strongest Invariant Assertion

We consider a while loop $w = \{\texttt{while t do b}\}$ on space $S$ and we let $W$ be its function. We seek to characterize the invariant assertion that is sufficiently strong the prove the strongest possible claim about the loop, namely (for a given state $s_0$):

$$\{s = s_0\}\ \texttt{while t do b}\ \{s = W(s_0)\}.$$

We had proven, in section 4.2.1, that $WC$ is an invariant assertion (i.e. satisfies the second verification condition of Hoare's rule) for any vector $C$ on $S$; in this section, we instantiate $C$ as the singleton $\{(s, s') | s = W(s_0)\}$; whence the vector $WC$ can be written as $WC = \{(s, s') | W(s) = W(s_0)\}$. The remaining verification conditions of Hoare's rule are then checked, as follows:

- First condition: $s = s_0 \Rightarrow W(s) = W(s_0)$.
- Third condition:

$$\begin{array}{ll} & W(s) = W(s_0) \wedge \neg t(s) \\ \Rightarrow & \quad \{\text{ Mills' Theorem }\} \\ & W(s) = W(s_0) \wedge W(s) = s \\ \Rightarrow & \quad \{\text{ Logic }\} \\ & s = W(s_0). \end{array}$$

As an illustration, we consider the sample loop introduced in section 3, and we let $s_0$ be the state defined by $n(s_0) = n_0$, $k(s_0) = 1$, $f(s_0) = 1$. Also, we write the function of this while loop as:

$$W\begin{pmatrix} n \\ f \\ k \end{pmatrix} = \begin{pmatrix} n \\ f \times \frac{n!}{(k-1)!} \\ n+1 \end{pmatrix}.$$

Then, $W(s) = W(s_0)$ can be written as:

$$n = n_0 \wedge f \times \frac{n!}{(k-1)!} = 1 \times \frac{n_0!}{0!} \wedge n + 1 = n_0 + 1,$$

which can be simplified to

$$n = n_0 \wedge f = (k-1)!.$$

This is indeed an adequate invariant assertion for the given precondition and postcondition.

#### 4.4.2 Strongest Invariant Relation

We have seen in Proposition 7 (section 4.2) that invariant relations can be used to approximate the function of the loop. For this purpose, an adequate invariant relation is one that enables us to actually compute (rather than simply approximate) the function of the loop. We propose the following candidate:

$$R = (T \cap B)^*$$

and we prove that it is indeed an invariant relation; then we check that it produces the function of the loop, by virtue of the formula of Proposition 7. This relation is reflexive and transitive, by construction. Also, it is, by construction, a superset of $(T \cap B)$. According to Proposition 7, the following relation is a lower bound for $W$:

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

Yet, according to Theorem 2, this is precisely equal to function $W$, suggesting that the invariant relation we have chosen is adequate. As an illustration, we consider the sample loop introduced in section 3, and we write:

$$\begin{array}{ll} & T \cap B \\ = & \qquad \{ \text{ substitutions } \} \\ & \{(s, s') | k \neq n+1 \wedge n' = n \wedge f' = f \times k \wedge k' = k+1\}. \end{array}$$

We claim that the transitive closure of this relation is:

$$R = \{(s, s') | n' = n \wedge \frac{f}{(k-1)!} = \frac{f'}{(k'-1)!} \wedge k \leq k'\}.$$

As we recall, the reflexive transitive closure of a relation is the smallest reflexive transitive superset of that relation. We content ourselves, in this discussion, with observing that it is a reflexive and transitive superset of $(T \cap B)$, but not that it is minimal. Reflexivity and transivity can be inferred

| Criterion | Invariant Assertion | Invariant Function | Invariant Relation |
|---|---|---|---|
| Name, Attribute | $A$, vector | $V$, total deterministic | $R$, reflexive transitive |
| Condition | $A \cap T \cap B \subseteq \widehat{A}$ | $T \cap BV \subseteq V$ | $(T \cap B) \subseteq R$ |
| Example | $A = \{(s,s') \mid f = (k-1)!\}$ | $V \begin{pmatrix} n \\ f \\ k \end{pmatrix} = \begin{pmatrix} 0 \\ \frac{f}{(k-1)!} \\ 0 \end{pmatrix}$ | $R = \{(s,s') \mid \frac{f}{(k-1)!} = \frac{f'}{(k'-1)!}\}$ |
| Supersets of $(T \cap B)$ | $(\overline{A} \cup \widehat{A})$ | $(V\widehat{V})$ | $R$ |
| Relation to Loop Function | $A = WC$ for arbitrary vector $C$ | $V = W$ | $R = W\widehat{W}$ |
| Ordering | Implication Inclusion | Injectivity | Refinement |
| Weakest | **true** | Constant functions | $L$ |
| Optimal / Adequate Invariants | $A = \{(s,s') \mid W(s) = W(s_0)\}$ for some $s_0 \in S$ | $V = W$ | $R = (T \cap B)^*$ |

Figure 1: Characterizing Invariants

by inspection; as for the condition that $R$ is a superset of $(T \cap B)$, we can establish it readily by observing that the intersection $R \cap (T \cap B)$ can be shown to equal $(T \cap B)$. By virtue of Proposition 7, we derive the following lower bound for $W$,

$$
\begin{array}{ll}
 & R \cap \widehat{\overline{T}} \\
= & \quad \{ \text{ substitutions } \} \\
 & \{(s,s') \mid n' = n \wedge \frac{f}{(k-1)!} = \frac{f'}{(k'-1)!} \wedge k \leq k' \wedge k' = n'+1\} \\
= & \quad \{ \text{ simplification } \} \\
 & \{(s,s') \mid k \leq n+1 \wedge n' = n \wedge f' = n! \times \frac{f}{(k-1)!} \wedge k' = n+1\} \\
= & \quad \{ \text{ since } k \leq n+1 \text{ is a state condition, we do not have to write it } \} \\
 & \{(s,s') \mid n' = n \wedge f' = n! \times \frac{f}{(k-1)!} \wedge k' = n+1\}
\end{array}
$$

This is a total and deterministic relation, which is the actual function of the loop.

Summing up: Figure 1 compiles in tabular form the results we have discussed regarding the distinguishing characteristics of invariant assertions, invariant relations, and invariant functions.

## 5 Relations between Invariants

In this section, we analyze the inter-relationships between the three invariants.

### 5.1 Invariant Assertions and Invariant Relations

The first question that we raise in this section is: can an invariant relation be used to generate an invariant assertion? The answer is provided by the following proposition.

**Proposition 9** *Let $R$ be an invariant relation of $w = \{\texttt{while t do b}\}$ on space $S$ and let $C$ be an arbitrary vector on $S$. Then $\widehat{R}C$ is an invariant assertion for $w$.*

**Proof.** Relation $\widehat{R}C$ is a vector since $C$ is a vector. We must prove $\widehat{R}C \cap T \cap B \subseteq \widehat{\widehat{R}C}$. To do so, we proceed as follows:

$$
\begin{array}{ll}
 & \widehat{R}C \cap T \cap B \subseteq \widehat{\widehat{R}C} \\
\Leftarrow & \qquad \{ \text{ Since } T \cap B \subseteq R \ \} \\
 & \widehat{R}C \cap R \subseteq \widehat{C}R \\
\Leftarrow & \qquad \{ \text{ since } \widehat{R}C \cap R \subseteq L(\widehat{R}C \cap R) \ \} \\
 & L(\widehat{R}C \cap R) \subseteq \widehat{C}R \\
\Leftrightarrow & \qquad \{ \text{ vector identity 4 } \} \\
 & \widehat{C}RR \subseteq \widehat{C}R \\
\Leftarrow & \qquad \{ \text{ monotonicity } \} \\
 & RR \subseteq R,
\end{array}
$$

which holds by virtue of the transitivity of $R$. **qed**

This proposition is interesting to the extent that it shows how to derive an invariant assertion from an invariant relation, but also because it shows that an invariant relation can generate (potentially) an infinity of invariant assertions, one for each vector $C$. We consider the following example, pertaining to the sample loop introduced in section 3, where we take an invariant relation

$$R = \{(s, s') | \tfrac{f}{(k-1)!} = \tfrac{f'}{(k'-1)!}\},$$

and a set of vectors, say

$$
\begin{aligned}
C_0 &= \{(s, s') | f(s) = 1 \wedge k(s) = 1\} \\
C_1 &= \{(s, s') | f(s) = 1 \wedge k(s) = 2\} \\
C_2 &= \{(s, s') | f(s) = 2 \wedge k(s) = 3\} \\
C_3 &= \{(s, s') | f(s) = 6 \wedge k(s) = 4\} \\
C_4 &= \{(s, s') | f = (k-1)!\} \\
C_5 &= \{(s, s') | f(s) = 2 \wedge k(s) = 5\}.
\end{aligned}
$$

The invariant assertion that stems from vector $C_0$ is:

$$A = \{(s, s') | \exists s'' : \frac{f}{(k-1)!} = \frac{f"}{(k"-1)!} \wedge f" = 1 \wedge k" = 1\},$$

which can be simplified to

$$A = \{(s, s') | f = (k-1)!\}.$$

We leave it to the reader to check that the invariant assertions derived from vectors $C_1$, $C_2$, $C_3$ and $C_4$ are the same as the relation given above, since in all cases we have $\frac{f"}{(k"-1)!} = 1$. The invariant assertion that stems from vector $C_5$ is:

$$A_5 = \{(s, s')|\exists s'' : \frac{f}{(k-1)!} = \frac{f"}{(k"-1)!} \wedge f" = 2 \wedge k" = 5\},$$

which can be simplified to:

$$A = \{(s, s')|\frac{f}{(k-1)!} = \frac{1}{12}\},$$

or to

$$A = \{(s, s')|f = \frac{(k-1)!}{12}\}.$$

Through this example, we want to illustrate the idea that in the formula of invariant assertion provided by Proposition 9, the term $\widehat{R}$ pertains to the while loop alone, whereas $C$ pertains to the context in which the loop is placed, viz its initalization. So that if the same loop is used with different initializations, we change $C$ but maintain $R$.

We now consider the question of generating an invariant relation from an invariant assertion, for which we have the following proposition.

**Proposition 10** *Given an invariant assertion $A$ for while loop $w = \{$`while to do b`$\}$ on space $S$, the relation $R = \overline{A} \cup \widehat{A}$ is an invariant relation for $w$.*

**Proof.** Proposition 3 provides that $R$ thus defined is a reflexive and transitive superset of $(T \cap B)$, hence it satisfies the conditions of Definition 6. **qed**

Now that we find that we can derive an invariant assertion from any invariant relation and an invariant relation from any invariant assertion, we need to ask the following questions: can any invariant assertion be derived from an invariant relation, and can any invariant relation be derived from an invariant assertion? The answers are provided below.

**Proposition 11** *Given an invariant assertion $A$, there exists an invariant relation $R$ and a vector $C$ such that $A = \widehat{R}C$.*

**Proof.** If $A$ is empty then this proposition holds vacuously for $C = \emptyset$. Given a non-empty invariant assertion $A$, we let $R = \overline{A} \cup \widehat{A}$ and $C = A$, and we prove that $A = \widehat{R}C$; we already know, by Proposition 10 that $R$ is an invariant relation. What remains to prove:

$$\begin{aligned}
& \widehat{R}C \\
= & \quad \{ \text{ substitutions } \} \\
& \widehat{(\overline{A} \cup \widehat{A})}A \\
= & \quad \{ \text{ distributing the converse operation } \} \\
& (\widehat{\overline{A}} \cup A)A \\
= & \quad \{ \text{ distributivity } \} \\
& \widehat{\overline{A}}A \cup AA \\
= & \quad \{ \text{ identity 6 } \}
\end{aligned}$$

$$
\begin{array}{ll}
& AA \\
= & \quad \{ \text{ definition of a vector } \} \\
& ALAL \\
= & \quad \{ \text{ associativity, and identity 14 } (A \neq \emptyset) \} \\
& AL \\
= & \quad \{ \text{ definition of a vector } \} \\
& A.
\end{array}
$$

**qed**

As to the matter of whether any invariant relation can be generated from an invariant assertion, the answer appears to be no, though we have a substitute: any invariant relation can be generated as an intersection of elementary invariant assertions. To formulate this result, we recall the concept of *point*, which is a special type of vector. While a vector is defined by a subset of $S$, a point is defined by a singleton. As an illustration, we consider the set $S$ defined by natural variables $n$, $f$ and $k$, and we write a few vectors and a few points, to illustrate the distinction.

$$
\begin{array}{l}
C_0 = \{(s, s') | f = 1\}, \\
C_1 = \{(s, s') | f = (k-1)!\}, \\
C_2 = \{(s, s') | f = 1 \wedge k = 1\}, \\
p_0 = \{(s, s') | n = 6 \wedge f = 1 \wedge k = 1\}, \\
p_1 = \{(s, s') | n = 6 \wedge f = 120 \wedge k = 7\}, \\
p_2 = \{(s, s') | n = 9 \wedge f = 2 \wedge k = 5\}.
\end{array}
$$

We have the following proposition.

**Proposition 12** *Given an invariant relation $R$, we can write $R$ as:*

$$R = \left( \bigcap_{p \in S} (\overline{A} \cup \widehat{A}) \right),$$

*where $A = \widehat{R}p$.*

**Proof.** We prove a lemma, to the effect that for an arbitrary vector $C$ and a reflexive transitive relation $R$ the following equation holds:

(23) $\widehat{R}C \cup \widehat{\overline{CR}} \cup R = L.$

The proof of this lemma proceeds as follows:

$$
\begin{array}{ll}
& \widehat{R}C \cup \widehat{\overline{CR}} \cup R = L \\
= & \quad \{ \text{ Identity (4) } \} \\
& \widehat{R}C \cup L(\overline{C} \cap \overline{R}) \cup R
\end{array}
$$

$$
\begin{array}{ll}
\supseteq & \{ \text{ reflexivity of } R, \text{ Boolean algebra, } I \subseteq L \ \} \\
& C \cup (\overline{C} \cap \overline{R}) \cup (\overline{C} \cap R) \\
= & \{ \text{ set theory } \} \\
& L.
\end{array}
$$

Because $L$ is maximal, $L \subseteq \widehat{R}p \cup \widehat{\overline{p}}\overline{R} \cup R$ is equivalent to $L = \widehat{R}p \cup \widehat{\overline{p}}\overline{R} \cup R$.

In order to prove the equality given in this proposition, we prove inclusion in both directions.

- Proof of $R \subseteq \left( \bigcap_{p \in S} (\overline{A} \cup \widehat{A}) \right)$. This formula is a logical consequence of:

$$\forall p \in S : R \subseteq \overline{\widehat{R}p} \cup \widehat{p}R,$$

which we prove as follows:

$$
\begin{array}{ll}
& R \subseteq \overline{\widehat{R}p} \cup \widehat{p}R \\
\Leftrightarrow & \{ \text{ set theory } \} \\
& \widehat{R}p \cap R \subseteq \widehat{p}R \\
\Leftrightarrow & \{ \text{ identity 12 } \} \\
& L(\widehat{R}p \cap R) \subseteq \widehat{p}R \\
\Leftrightarrow & \{ \text{ identity 4) } \} \\
& \widehat{p}RR \subseteq \widehat{p}R \\
\Leftarrow & \{ \text{ monotonicity of product } \} \\
& RR \subseteq R \\
\Leftrightarrow & \{ \text{ transitivity of } R \ \} \\
& \textbf{true} \ .
\end{array}
$$

- Proof of $\left( \bigcap_{p \in S} (\overline{A} \cup \widehat{A}) \right) \subseteq R$. In appendix B, we prove the following equality:

$$\left( \bigcap_{p \in S} \overline{\widehat{R}p} \cup \widehat{p}R \right) = \left( \bigcup_{X \subseteq S} \left( \bigcap_{s \in X} \overline{\widehat{R}p} \right) \cap \left( \bigcap_{s \in \overline{X}} \widehat{p}R \right) \right).$$

By virtue of this equality, it suffices for us to prove that for all subset $X$ of $S$, we have

$$\left( \bigcap_{s \in X} \overline{\widehat{R}p} \right) \cap \left( \bigcap_{s \in \overline{X}} \widehat{p}R \right) \subseteq R.$$

To this effect, we proceed as follows:

$$
\begin{array}{ll}
& \left( \bigcap_{s \in X} \overline{\widehat{R}p} \right) \cap \left( \bigcap_{s \in \overline{X}} \widehat{p}R \right) \subseteq R \\
\Leftrightarrow & \{ \text{ Boolean Algebra } \} \\
& L \subseteq \left( \bigcup_{s \in X} \widehat{R}p \right) \cup \left( \bigcup_{s \in \overline{X}} \overline{\widehat{p}R} \right) \cup R
\end{array}
$$

$\Leftrightarrow$ { Identity: 16 }

$L \subseteq \left(\bigcup_{s\in X} \widehat{R}p\right) \cup \left(\bigcup_{s\in \overline{X}} \widehat{p}\overline{R}\right) \cup R$

$\Leftrightarrow$ { Distributing $\circ$ over $\cup$ }

$L \subseteq \widehat{R}\left(\bigcup_{s\in X} p\right) \cup \left(\bigcup_{s\in \overline{X}} \widehat{p}\right) \overline{R} \cup R$

$\Leftrightarrow$ { Distributing the inverse over $\cup$ }

$L \subseteq \widehat{R}\left(\bigcup_{s\in X} p\right) \cup \widehat{\left(\bigcup_{s\in \overline{X}} p\right)}\overline{R} \cup R$

$\Leftrightarrow$ { if we let $C$ be the vector: $\left(\bigcup_{s\in X} p\right)$ }

$L \subseteq \widehat{R}C \cup \widehat{\overline{C}}\overline{R} \cup R$

$\Leftrightarrow$ { Lemma above }

**true** .

**qed**

As an illustration of this Proposition, we consider the factorial loop presented in section 3, and let $R$ be the invariant relation defined by:

$$R = \left\{(s, s')|\frac{f}{(k-1)!} = \frac{f'}{(k'-1)!}\right\},$$

and we let $p$ be the point defined by

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

Then,

$A$

$=$ { proposed formula }

$\widehat{R}p$

$=$ { $R$ is symmetric }

$Rp$

$=$ { substitution }

$\{(s, s')|\frac{f}{(k-1)!} = \frac{f'}{(k'-1)!}\} \circ \{(s, s')|s = p\}$

$=$ { relational product }

$\{(s, s')|\frac{f}{(k-1)!} = \frac{f(p)}{(k(p)-1)!}\}.$

From which we infer:

$\overline{A} \cup \widehat{A}$

$=$ { interpretation, substitution }

$\{(s, s')|\frac{f}{(k-1)!} = \frac{f(p)}{(k(p)-1)!} \Rightarrow \frac{f'}{(k'-1)!} = \frac{f(p)}{(k(p)-1)!}\}.$

Taking the intersection of all such relations, for all $p$ in $S$, we find:

$$
\begin{array}{ll}
 & \bigcap_{p\in S}\{(s,s')|\frac{f}{(k-1)!}=\frac{f(p)}{(k(p)-1)!}\Rightarrow\frac{f'}{(k'-1)!}=\frac{f(p)}{(k(p)-1)!}\} \\
= & \quad \{ \text{ interpretation } \} \\
 & \{(s,s')|\forall p:\frac{f}{(k-1)!}=\frac{f(p)}{(k(p)-1)!}\Rightarrow\frac{f'}{(k'-1)!}=\frac{f(p)}{(k(p)-1)!}\} \\
= & \quad \{ \text{ simplification } \} \\
 & \{(s,s')|\frac{f}{(k-1)!}=\frac{f'}{(k'-1)!}\} \\
= & \quad \{ \text{ substitution } \} \\
 & R.
\end{array}
$$

To conclude, from an invariant relation, we can derive as many invariant assertions as there are vectors on $S$ (infinitely many, if $S$ is infinite); and it takes a large number (possibly infinity) of invariant assertions (one for each element of $S$) to produce an invariant relation; furthermore, any invariant assertion stems from an invariant relation.

## 5.2 Invariant Relations and Invariant Functions

With regards to the relationship between invariant functions and invariant relations, we have the following propositions.

**Proposition 13** *Let $V$ be an invariant function for while statement $w = \{$`while t do b`$\}$. Then the nucleus of $V$ is an invariant relation for $w$.*

**Proof.** This proposition stems readily from Proposition 4 and from the definition of invariant functions. **qed**

As to the question of whether any invariant relation is the nucleus of an invariant function, we answer in the negative, as shown by the following example: if we consider the sample loop of section 3, and we let $R$ be defined by

$$R = \{(s.s')|k \leq k'\},$$

then this relation is clearly reflexive and transitive; in addition, it is a superset of the function of the loop body. Yet, it is not the nucleus of any function, because a nucleus is symmetric and this relation is not. This example inspires us the following proposition.

**Proposition 14** *Let $R$ be a symmetric invariant relation for $w = \{$`while t do b`$\}$. Then there exists an invariant function $V$ whose nucleus equals $R$.*

**Proof.** Since $R$ is an equivalence relation, we let $V$ be the function that to each element of $S$ associates its equivalence class modulo $R$ (this function can be identified with the *power transpose* of relation $R$ [2]). This function is total, since $R$ is reflexive. We analyze its nucleus:

$$
\begin{array}{ll}
 & V\widehat{V} \\
= & \quad \{ \text{ definition, where } s.R \text{ is the set of images of } s \text{ by } R \text{ } \} \\
 & \{(s,s')|s.R = s'.R\} \\
\subseteq & \quad \{ \text{ because } R \text{ is total, both } s.R \text{ and } s'.R \text{ are non-empty } \} \\
 & \{(s,s')|(s,s') \in R\widehat{R}\}
\end{array}
$$

$$
\begin{array}{ll}
= & \{ \text{ notation } \} \\
\quad R\widehat{R} & \\
\subseteq & \{ \text{ symmetry, transitivity of } R \} \\
\quad R. &
\end{array}
$$

In addition, given $(s, s')$ in $R$, we know by definition that $s$ and $s'$ have the same equivalence class by $R$, hence $V(s) = V(s')$. Hence $(s, s') \in V\widehat{V}$. We conclude: $R = V\widehat{V}$.

What we must prove now is that $V$ is an invariant function. This is straightforward, starting from the hypothesis that $R$ is an invariant relation:

$$
\begin{array}{ll}
\quad (T \cap B) \subseteq R & \\
\Leftrightarrow & \{ \text{ substitution } R = V\widehat{V} \ \} \\
\quad (T \cap B) \subseteq V\widehat{V} & \\
\Leftrightarrow & \{ \text{ identity 17 } \} \\
\quad (T \cap B)V \subseteq V. &
\end{array}
$$

**qed**

As an illustration, we consider the following symmetric invariant relation of the sample loop,

$$R = \{(s, s') | \frac{f}{(k-1)!} = \frac{f'}{(k'-1)!}\},$$

and we put forth the following functions as possible invariant functions whose nucleus is relation $R$.

$$V_0 \begin{pmatrix} n \\ f \\ k \end{pmatrix} = \begin{pmatrix} 0 \\ \frac{f}{(k-1)!} \\ 0 \end{pmatrix}, \ V_1 \begin{pmatrix} n \\ f \\ k \end{pmatrix} = \begin{pmatrix} \frac{f}{(k-1)!} \\ 2 \\ 4 \end{pmatrix}, \ V_2 \begin{pmatrix} n \\ f \\ k \end{pmatrix} = \begin{pmatrix} 4 \\ 2 \times \frac{f}{(k-1)!} \\ 5 \times \frac{f}{(k-1)!} \end{pmatrix}.$$

Even though not all invariant relations stem from invariant functions, we must make the following qualifications, borne out by several years' worth of observations:

- Virtually all useful invariant relations have a symmetric component, i.e. are the intersection of a symmetric invariant relation with an antisymmetric invariant relation (an invariant relation cannot be asymmetric since it is reflexive).
- The symmetric component of an invariant relation typically carries a great deal of invariant information on the loop.

In practice, invariant functions are our most useful source of functional information about a while loop.

### 5.3 Invariant Assertions and Invariant Functions

In section 4.2, we have seen that if $W$ is the function of the loop, then $WC$ is an invariant assertion of the loop, for any vector $C$ on $S$. This is in fact valid for any invariant function, not only for the loop function.

| Criterion | Invariant Assertion | Invariant Function | Invariant Relation | Loop Function |
|---|---|---|---|---|
| Invariant Assertion | Is | No relation was found | $R = \overline{A} \cup \widehat{A}$ | $W \sqsupseteq (\overline{A} \cup \widehat{A}) \cap \widehat{\overline{T}}$ |
| Invariant Function | $A = VC$ | Is | $R = V\widehat{V}$ | $W \sqsupseteq (V\widehat{V}) \cap \widehat{\overline{T}}$ |
| Invariant Relation | $A = \widehat{R}C$ | if $R = \widehat{R}$ $V : S \to S/R$ | Is | $W \sqsupseteq R \cap \widehat{\overline{T}}$ |
| Loop Function | $A = WC$ | IsA | $R = W\widehat{W}$ | Is |

Figure 2: Relations Between Invariants

**Proposition 15** *If $V$ is an invariant function for $w = \{$`while t do b`$\}$, which terminates for all elements of its space $S$, then $A = VC$ is an invariant assertion for $w$, for any vector $C$ on $S$.*

**Proof.** This result stems readily from Definitions 1 and 2, and from Proposition 1. **qed**

As an illustration, if we take the invariant function $V_0$ presented in the previous section and for $C$ the point $C = \{(s, s') | s = \langle 0, 1, 0 \rangle\}$, we find the following invariant assertion:

$$A = \{(s, s') | f = (k-1)!\}.$$

If we choose the same invariant function but the point $C' = \{(s, s') | s = \langle 1, 1, 1 \rangle\}$, we find the invariant assertion $A = \emptyset$. To conclude this section on the relation between invariant assertions and invariant functions, we wish to point out what is *not* a relation between these concepts: An invariant assertion is not a Boolean valued invariant function, since an invariant assertion may be false before execution of the loop body and become true after, while an invariant function takes the same values before and after execution of the loop body. This leads us to suggest that a better name for invariant assertions is *monotonic* assertions.

### 5.4 A Tabular Synopsis

Table 2 shows in succint tabular form the main relations that exist between invariant assertions, invariant relations and invariant functions. We have added to the comparison the function of the loop, to highlight the bilateral relations between loop functions and each one of the invariants. In this table, each entry gives, when possible, an expression that provides the artifact of the column (or an approximation thereof) as a function of the artifact of the row. We represent invariant assertions by $A$, invariant functions by $V$, invariant relations by $R$ and loop functions by $W$. Also, we let $C$ be an arbitrary subset of $S$, represented by a vector on $S$. Note the distinction between labels *Is* and *IsA*.

# 6 Computing Loop Functions

In section 4.2.2, we presented a Proposition to the effect that invariant relations can be used to approximate the function of the loop; in this section, we further elaborate on that result, by showing how, with a sufficient number of sufficiently refined invariant relations, we can capture the function of the loop, even for large and non-trivial loops. The detailed algorithm that we use to this effect, as well as the mathematics that form its foundations, are beyond the scope of our paper, and are given in [51]. The key ideas that underpin our algorithm are the following:

- Under the hypotheses of our study, we are interested in while loops written in a (deterministic) C-like programming language that terminate for all initial states; from this, we infer that $W$ is total and deterministic. Hence, $W$ is maximal in the refinement lattice; consequently, we can derive $W$ by successive approximations in the form of lower bounds.

- Proposition 7 provides a formula for computing lower bounds for $W$ from invariant relations of the while loop. Hence we can generate lower bounds for $W$ if we find (a sufficient number of sufficiently refined) invariant relations of $w$. By lattice theory, the lower bounds $Y_1$, $Y_2$, ... $Y_k$ can be combined by the lattice operator of join to produce the function of the loop, or an approximation thereof.

- Because invariant relations are supersets of the loop body's function, we can prepare the loop for the extraction of invariant relations by writing its function as an intersection of terms; once it is written as an intersection, any superset of any term or combination of terms is a superset of the function of the loop body.

- When the loop body includes if-then-else statements, the outer structure of its function is a union rather than an intersection; in that case, we find a superset for each term of the union, then we merge them using a specially programmed function. The role of this function is to take several reflexive transitive relations (which represent the invariant relations corresponding to each branch of the loop body) and find a (preferably the smallest) reflexive transitive superset thereof (while the union of reflexive relations is reflexive, the union of transitive relations is not necessarily transitive).

- The invariant relations of individual branches are generated by pattern matching, using patterns that are developed off-line by means of invariant functions. These patterns, which we call *recognizers*, capture all the programming knowledge and domain knowledge that is required to compute loop functions in usable form.

- The lower bounds are represented as equations between initial states and final states of the loop, and are generated in the syntax of Mathematica (©Wolfram Research). When these equations are submitted to Mathematica to solve in the final (primed) values of program variables as a function of initial values, we obtain the function of the loop or (if we have not generated a sufficient number of invariant relations/ lower bounds, or have not merged them properly) an approximation of the loop function.

Hence the function of the loop is derived from the source code in a four-phase process:

1. **cpp2cca**: The first step is to transform the source code (.cpp) onto a form that represents the function of the loop body as an intersection, or as a union of intersections. The notation

we use for this purpose is called *conditional concurrent assignments*, or CCA for short. This step is carried out by a complier, which we have developed using simple compiler generation technology.

2. **cca2mat**: Using the database of recognizers, we search for patterns in the CCA code, for which we have a corresponding pattern of an invariant relation; whenever a match is successful, we generate the corresponding invariant relation, which we represent as a set of Mathematica equations between initial states and final states. This step is carried out by a C++ program that converts CCA code into Mathematica equations, involving unprimed program variables (representing initial states) and primed program variables (representing final states). In its current version, this program proceeds by performing a syntactic match of the source code against code patterns of pre-stored recognizers. We are currently working on a more effective version based on semantic matching; the principle is the same but semantic matching enables us to do more with fewer, more generic, recognizers.

3. **merger**. If the loop body has a single branch (no if-then-else or if-then statements) then we skip this step; if not, we merge the invariant relations corresponding to the different branches of the loop body into a single invariant relation. This step is carried out by a special-purpose program that we have written in Mathematica; it proceeds by logical manipulations to compute a transitive superset of the union of transitive relations.

4. **mat2nb**. The equations stemming from the lower bounds are submitted to Mathematica for resolution; they are solved in the output values as a function of the input values, yielding an explicit expression of the function of the loop.

For the sake of illustration, we show below a sample C++ program, along with the function computed by our system for this program. Due to space limitation, we only present the source code and the final function representation for this program; interested readers may consult

`http://web.njit.edu/ mili/sampleloop.pdf`,

where we show the successive files that are generated for this program (loop.cpp, loop.cca, loop.mat, loop.mrg, loop.nb), as well as the recognizers that are invoked to analyze this loop.

The source program in C++:

```
#include <iostream>
#include <list>
#include<math.h>
using namespace std;
int Fact (int z);  // the factorial function
int f();           //  an arbitrary function
int main()
{
const int ca,cb,cd,ce,cN;  int i,j,k,h,y,m,q,w,x2,fx;
float ma,st,ut,x1,t,p,n,g,r,s,u,v,z,ta,ka,la,uv;
list <int> l1,l2; float aa[]; float ab[];
while (l2.size()!=0)
   {r=pow(i,5) + r; s=s+2*u; k=ca*h+k; la=pow(x1,j)/Fact(j) + la;
```

```
    l1.push_back(l2.front()); h=h+j; m=m+1;  j=j+i; fx=f(fx);
    g=g-15*cd; q= 1+2*i + q; ma=ka-ma; i=i+1; st=st+aa[i]; j= j-i;
    w=4*w; ut=ut+ab[j]; ma=(cd+1)*ka - ma; ka=ka-1; ta=pow(ta,3);
    if(x2%4==0)
       {x2=x2/4; y= y+2; t= t*4;}
    else
       {if (x2%2==0) {x2= x2/2; y= y+1; t= t*2;}
        else {x2= x2-1; z=z+t;}}
    ka=3+3*ka; w= cd+ w/2; p= 2*pow(p,3); m=2*m -2; n=1 + n/2;
    s=(cb-2)*u + s; h=h-1+cb-j; g=3*cd + g/5; v=pow(v,4); u=ca+u;
    uv=pow(uv,5); l2.pop_front();}
}
```

The function of this program (where $\Gamma$ is Euler's Gamma function):

$$
W \sqsupseteq \left\{ (s,s') \middle| \begin{array}{l}
aa' = aa \wedge ab' = ab \wedge fx' = f^{Size[l2]}(fx) \wedge g' = 5^{-Size[l2]} \times g \wedge \\
h' = h + cb \times Size[l2] \wedge i' = i + Size[l2] \wedge j' = j - Size[l2] \wedge ka' = 3^{Size[l2]} \times ka \wedge \\
k' = \dfrac{(2 \times k - ca \times cb \times Size[l2] + 2 \times ca \times h \times Size[l2] + ca \times cb \times Size[l2]^2)}{2} \wedge \\
l1' = Concat[l1, l2] \wedge l2' = \epsilon \wedge \\
la' = \dfrac{(la\Gamma[1+j]\Gamma[1+j-Size[l2]] + e^{x1}\Gamma[1+j-Size[l2]]\Gamma[1+j,x1]}{(\Gamma[1+j] \times \Gamma[1+j-Size[l2]])} \\
\quad - \dfrac{e^{x1} \times \Gamma[1+j]\Gamma[1+j-Size[l2],x1])}{(\Gamma[1+j] \times \Gamma[1+j-Size[l2]])} \wedge \\
m' = 2^{Size[l2]} \times m \wedge ma' = \dfrac{(-cd \times ka + 3^{Size[l2]} \times cd \times ka + 2 \times ma)}{2} \wedge \\
n' = 2^{(-Size[l2])}(-2 + 2^{(1+Size[l2])} + n) \wedge p' = 2^{(-\frac{1}{2} + \frac{3^{(Size[l2])}}{2})} p^{(3^{(Size[l2])})} \wedge \\
q' = q + 2 \times i \times Size[l2] + Size[l2]^2 \wedge \\
r' = \dfrac{1}{12} \times 12 \times r - 2 \times i \times Size[l2] + 20 \times i^3 \times Size[l2] - 30 \times i^4 \times Size[l2] \\
\quad + 12 \times i^5 \times Size[l2] - Size[l2]^2 + 30 \times i^2 \times Size[l2]^2 - 60 \times i^3 \times Size[l2]^2 \\
\quad + 30 \times i^4 \times Size[l2]^2 + 20 \times i \times Size[l2]^3 - 60 \times i^2 \times Size[l2]^3 \\
\quad + 5 \times Size[l2]^4 - 30 \times i \times Size[l2]^4 + 30 \times i^2 \times Size[l2]^4 - 6 \times Size[l2]^5 \\
\quad + 40 \times i^3 \times Size[l2]^3 + 12 \times i \times Size[l2]^5 + 2 \times Size[l2]^6) \wedge \\
s' = \dfrac{(2 \times s - ca \times cb \times Size[l2] + 2 \times cb \times u \times Size[l2] + ca \times cb \times Size[l2]^2)}{2} \wedge \\
st' = st + \displaystyle\sum_{index=1+i}^{i+Size[l2]} aa[index] \wedge t' = 2^{(Floor[\frac{Log[x2]}{Log[2]}] - Floor[\frac{Log[x2']}{Log[2]}])} t \wedge \\
ta' = ta^{3^{Size[l2]}} \wedge u' = u + ca \times Size[l2] \wedge ut' = ut + \displaystyle\sum_{index=j-Size[l2]}^{-1+j} ab[index] \wedge \\
uv' = uv^{5^{Size[l2]}} \wedge v' = v^{2^{(2Size[l2])}} \wedge w' = -cd + 2^{Size[l2]} \times cd + 2^{Size[l2]} \times w \wedge \\
x1' = x1 \wedge y' = y + Floor[\dfrac{Log[x2]}{Log[2]}] - Floor[\dfrac{Log[x2']}{Log[2]}] \wedge \\
z' = t \times x2 - 2^{(Floor[\frac{Log[x2]}{Log[2]}] - Floor[\frac{Log[x2']}{Log[2]}])} \times t \times x2' + z
\end{array} \right\}.
$$

The following comments help the reader understand the formula given above for the function of the loop.

- It may be helpful to look at the intermediate results that were generated in the process of computing the function of this loop; they are given in the file referenced above. This file shows the source code of this program (pages 1 and 2), followed by the CCA code (pages 2, 3, 4 and 5), followed by three sets of equations that correspond to the three branches of the CCA code/ the three execution paths of the source C++ code (pages 5, 7, 8 and 9), followed by the result of merging the non-common equations that appear in the three branches, as common equations are factored out before merge is called (page 9), followed by the result of

applying Mathematica to solve the merged equations in the primed variables as a function of the unprimed variables. Pages 6 and 7 contain a table that shows all the recognizers that were invoked in the analysis of this loop.

- The reader may notice that we have not been able to compute the final value of variable $x2$, hence strictly speaking what we have here is not the function of the loop, but an approximation (a fairly close approximation) of this function. We have determined that the reason for this shortcoming is that the code submitted to Mathematica is missing one equation. There are two possible remedies to this situation: either making some specific recognizers more generic, so that it is easier to merge them; or making the merge function more effective, so that it can generalize specific equations. This matter is currently under investigation, in the context of evolving our tool from syntactic matching to semantic matching.

- Note that we are able to handle a fairly large program using recognizers that deal with no more than three CCA statements at a time. If we represent CCA statements by nodes of a graph and invariant relations generated from patterns of 2 or 3 CCA statements by arcs between the statements in question, then we may be able to compute the function of the loop as soon as the graph defined by these arcs is connected.

- Note also that the loop body of this program computes a Taylor series; when the cca2mat recognizes a Taylor series, it replaces it by its closed form expression (in this case $e^{x1}$) and adjust with a residual term; this may explain the appearance of the Euler function ($\Gamma$) in the expression of $la'$.

# 7 Concluding Remarks

## 7.1 Summary

In this paper we have reviewed three distinct but related concepts, namely invariant assertions, invariant functions, and invariant relations. We have defined these concepts in relational terms, and have analyzed their various attributes, including: how they relate to the function of the loop, how they relate to the function of the loop body, how they are ordered, what is the weakest invariant of each type, and what is the strongest invariant of each type. Also, we analyzed the interrelationships among such invariants, by asking the questions: can we generate one type of invariant from another; can all invariants of one type be generated from invariants of another type. Among the main results of our study, we cite:

- All types of invariants involve searching a reflexive transitive superset of $(T \cap B)$; but while invariant assertions seek an inductive relation and invariant functions seek a symmetric relation, invariant relations impose no further conditions beyond reflexivity and transitivity.

- We can generate an invariant assertion from an invariant relation; furthermore, any invariant assertion can be generated from an invariant relation. In other words, if we focus exclusively on generating invariant relations, we necessarily cover all possible invariant assertions.

- Given a single invariant relation, we can generate a wide range of invariant assertions from it, typically infinitely many (one for each arbitrary vector on $S$). On the other hand, an arbitrary invariant relation cannot necessarily be generated from an invariant assertion, but

can be generated from a wide range of invariant assertions, typically infinitely many (one for each element of $S$).

- Given an invariant function, we can use it to generate an invariant relation; and any invariant function can be generated from an invariant relation. Hence, if we focus exclusively on generating invariant relations, we necessarily cover all possible invariant functions.

- Given a symmetric invariant relation, we can generate a wide range of invariant functions, typically infinitely many (one for each encoding of its equivalence classes). On the other hand, an invariant relation cannot necessarily be generated from an invariant function.

- Given an invariant relation, we can use it to generate lower bounds for the function of the loop in the lattice of refinement; with sufficient (and sufficiently refined) invariant relations, we can compute the function of the loop.

We have also summarily discussed the design and implementation of a tool that uses invariant functions and invariant relations to compute or approximate functions, and have illustrated it on a sample example.

Perhaps the simplest and most compelling result of our study is the finding (in Proposition 11) that *all* invariant assertions are the relational product of two components: the inverse of an invariant relation, which depends exclusively on the loop; the vector that represents the precondition of the loop, which depends exclusively on the context in which the loop is used. This insight may be used to streamline research on loop invariant generation [6, 8, 9, 15–17, 20, 21, 23, 24, 32–34, 37, 39, 40, 42, 43, 46, 59, 61, 64].

### 7.2 Research Prospects

This being to a large extent a survey paper, we are not envisioning extensions of it per se, but we will discuss extensions to our own work on invariant relations and invariant functions.

- *Lifting the Hypothesis $WL = L$.* This hypothesis has played an important role in the development of our theory, and has made many of our results possible; for example, assuming that $W$ is total and deterministic allows us to claim that $W$ is maximal in the refinement ordering, hence can be approximated using only lower bounds. But in practice it is difficult/ unnatural to separate the task of reasoning about the domain of $W$ from the task of reasoning about its expressions. Hence we envision to lift this hypothesis, to be able to reason about loops that may or may not terminate.

- *Functional Analysis of Loops.* In addition to computing or approximating loop functions, invariant relations can be used to generate invariant assertions (as we have discussed in this paper), to generate weakest preconditions and strongest postconditions [55], to generate termination conditions, and to verify the correctness of a loop with respect to a relational specification.

- *Generating Invariant Relations.* The applicability of our work is critically dependent on our ability to generate invariant relations from a static analysis of the source code of the loop. We are currently evolving our tool (especially the cca2mat component) by replacing its syntactic pattern matching with semantic pattern matching, by enriching its database of recognizers, and by specializing it to specific application domains.

We feel that these concepts offer complementary information on the functional properties of a while loop, and are worthy of further exploration.

### 7.3 Related Work

While we have isolated invariant relations as a worthy concept that carries intrinsic value, rather than being an auxiliary to computing invariant assertions, we lay no claim to being the first to have used them. We have encountered other works where invariant relations, or special forms thereof, are computed and used in the analysis of while loops. In [7], Carrette and Janicki derive properties of numeric iterative programs by modeling the iterative program as a recurrence relation, then solving these equations. Typically, the equations obtained from the recurrence relations by removing the recurrence variable are nothing but the reflexive transitive relations that we are talking about. However, whereas the method of Carrette and Janicki is applicable only to numeric programs, ours is applicable to arbitrary programs, provided we have adequate recognizers to match their control structure, and adequate axiomatizations of their data structure.

In [59], Podelski and Rybalchenko introduce the concept of *transition invariant*, which is a transitive (but not symmetric) superset of the loop body's function. Of special interest are transitive invariants which are well-founded (hence asymmetric); these are used to characterize termination properties or liveness properties of loops. Transitive invariants are related to invariant relations in the sense that they are both transitive supersets of the loop body. However, transition invariants and invariant relations differ on a very crucial attribute: whereas the latter are reflexive, and are used to capture what remains invariant from one iteration to the next, the former are asymmetric, hence not reflexive, and are used to capture what changes from one iteration to the next.

In [26], Kovacs et. al. deploy an algorithm (Aligator) they have written in Mathematica to generate invariant assertions of while loops using Cousot's Abstract Interpretation [13, 14]. They proceed by formulating then resolving the recurrence relations that are defined by the loop body, much in the same way (except for the automation) as Carette and Janicki advocate [7]; once they solve the recurrence relations, they obtain a binary relation beween the initial states and the *current* state, from which they derive an invariant assertion by imposing the pre-conditions on the initial state. Typically, this operation correspond to the formula that we propose in Proposition 9: the relation they find between initial and current states is an invariant relation ($R$); and initial conditions they dictate on the initial state correspond to vector $C$ of Proposition 9. Aligator does not take the converse of the invariant relation (as dictated by Theorem 9: $A = \widehat{R}C$), nor does it have to, since $R$ is typically symmetric.

In [24], Furia and Meyer present an algorithm for generating loop invariants by generalizing the postcondition of the loop. To this effect, they deploy a number of generalization techniques, such as constant relaxation (replacing a constant by a variable), uncoupling (replacing two occurrences of the same variable by different variables), term dropping (removing a conjunct), and variable aging (replacing a variable by an expression it had prior to termination). This work differs from ours in many ways, obviously, including that it does not distinguish between what is dependent on the loop and what is dependent on its context (hence must be redone for the same loop whenever the postcondition changes), that it is of a highly heuristic nature, and that it is highly syntactic (the same postcondition written differently may yield a different result). Another important distinction, of course, is that we analyze the loop as it is (without consideration of its specification), whereas Furian and Meyer analyze it as it should be (assuming it is correct).

# A Appendix: Invariant Assertions in Context

Throughout the paper, we have used a definition of invariant assertion that is dependent on the loop but not on its context (specifically, the precondition and postcondition that frame it). The following corollary shows how we can use the result of Proposition 9 to produce an invariant assertion that is adequate for a proof of correctness with respect to a precondition/ postcondition pair.

**Corollary 1** *We consider the while loop* $w = \{\texttt{while t do b}\}$*, and the precondition /postcondition specification* $(\phi, \psi)$*, which we represent relationally by the pair of vectors* $(\Phi, \Psi)$*. If* $R$ *is an invariant relation of* $w$ *such that* $\widehat{R}\Phi \cap \overline{T} \subseteq \Psi$*, then the following Hoare formula is valid:*

$$\{\phi\}\ w\ \{\psi\}.$$

**Proof.** We must identify an invariant assertion $\alpha$ (logical formula) that satisfies the following conditions:

1. $\phi \Rightarrow \alpha$,
2. $\{\alpha \wedge t\}\ b\ \{\alpha\}$,
3. $\alpha \wedge \neg t \Rightarrow \psi$.

If we represent each logical formula by its vector, we find the following relational equations:

1. $\Phi \subseteq A$,
2. $A \cap T \cap B \subseteq \widehat{A}$,
3. $A \cap \overline{T} \subseteq \Psi$.

If we take $A = \widehat{R}\Phi$, then the first condition holds by virtue of the reflexivity of $R$, the second condition holds by virtue of Proposition 9, and the third condition holds by hypothesis. **qed**

# B Proof of Lemma, Proposition 12

We must prove:

$$(\cap_{s\in S} Q_s \cup R_s) = (\cup_{X\subseteq S}(\cap_{s\in X} Q_s) \cap (\cap_{s\in \overline{X}} R_s)).$$

1. Proof of $\supseteq$. We have to prove that for all $X$ subset of $S$, we have

$$(\cap_{s\in X} Q_s) \cap (\cap_{s\in \overline{X}} R_s) \subseteq (\cap_{s\in S} Q_s \cup R_s).$$

$$\begin{array}{ll} & (\cap_{s\in X} Q_s) \cap (\cap_{s\in \overline{X}} R_s) \\ \subseteq & \qquad \{\text{ monotonicity }\} \\ & (\cap_{s\in X} Q_s \cup R_s) \cap (\cap_{s\in \overline{X}} Q_s \cup R_s) \\ = & \qquad \{\text{ associativity }\} \\ & (\cap_{s\in S} Q_s \cup R_s) \end{array}$$

2. Proof of $\subseteq$. We let $t$ be an element of $(\cap_{s\in S}Q_s \cup R_s)$ and we use a pointwise argument. To show that $t$ is in $(\cup_{X\subseteq S}(\cap_{s\in X}Q_s) \cap (\cap_{s\in\overline{X}}R_s))$ it suffices to find $X$ such that

$$t \in (\cap_{s\in X}Q_s) \cap (\cap_{s\in\overline{X}}.$$

We take $X = \{s|t \in Q_s\}$. We proceed as follows:

$$\begin{array}{ll} & t \in (\cap_{s\in X}Q_s) \cap (\cap_{s\in\overline{X}}R_s) \\ \Leftrightarrow & \qquad \{ \text{ definition of } X, \text{ logic } \} \\ & t \in (\cap_{t\in Q_s}Q_s) \wedge t \in (\cap_{t\notin Q_s}R_s) \\ \Leftrightarrow & \qquad \{ \text{ set theory } \} \\ & \textbf{true} \ \wedge t \in (\cap_{t\notin Q_s}R_s) \\ \Leftrightarrow & \qquad \{ \text{ from } t \in Q_s \cup R_s \text{ and } t \notin Q_s, \text{ we infer } t \in R_s \ \} \\ & \textbf{true} \ \wedge \textbf{true} \ . \end{array}$$

## Acknowledgement

The authors are very grateful to the anonymous reviewers for their courteous, constructive evaluation of this work, and very impressed by the truly outstanding quality of their reports. The reviewers' feedback has been a valuable resource that the authors have used to improve the form and content of this paper.

## References

[1] J. Berdine, A. Chawdhary, B. Cook, D. Distefano, and P. O'Hearn. Variance analyses from invariance analyses. In *Proceedings of the 34th Annual Symposium on Principles of Programming Languages*, Nice, France, 2007.

[2] R.S. Bird and O. DeMoor. *Algebra of Programming.* Prentice Hall International, 1997.

[3] A. Blikle and A. Mazurkiewicz. An algebraic approach to the theory of algorithms, languages and recursiveness. In *Proceedings, First Conference on Mathamatical Foundations of Computer Science*, Jablona, Poland, 1972.

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

[5] A. R. Bradley, Z. Manna, and H. B. Sipma. Termination analysis of integer linear loops. In *CONCUR*, pages 488–502, 2005.

[6] E. Rodriguez 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.

[7] J. Carette and R. Janicki. Computing properties of numeric iterative programs by symbolic computation. *Fundamentae Informatica*, 80(1-3):125–146, March 2007.

[8] 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.

[9] M. A. Colon, S. Sankaranarayana, and H. B. Sipma. 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.

[10] B. Cook, S. Gulwani, T. Lev-Ami, A. Rybalchenko, and M. Sagiv. Proving conditional termination. In *Proceedings of the 20th international conference on Computer Aided Verification*, CAV '08, pages 328–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[11] B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for system code. In *Proceedings, PLDI*, pages 415–426, 2006.

[12] B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '06, pages 415–426, New York, NY, USA, 2006. ACM.

[13] P. Cousot. Abstract interpretation. Technical Report www.di.ens.fr/c̃ousot/AI/, Ecole Normale Superieure, Paris, France, August 2008.

[14] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings, Fourth ACM Symposium on Principles of Programming Languages*, Los Angeles, CA, 1977.

[15] P. Cousot and R. Cousot. Automatic synthesis of optimal invariant assertions: Mathematical foundations. In *Proceeding Proceedings of the 1977 symposium on Artificial intelligence and programming languages*. ACM, 1977.

[16] 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.

[17] 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.

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

[19] H. Doornbos, R. Backhouse, and J. ven der Woude. A calculational approach to mathematical induction. *Theoretical Computer Science*, 179(1):103–135, 1997.

[20] 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.

[21] T. Fahringer and B. Scholz. *Advanced Symbolic Analysis for Compilers*. Springer Verlag, Berlin, Germany, 2003.

[22] R.W. Floyd. Assigning meaning to programs. *Proceedings of the American Mathematical Society Symposium in Applied mathematics*, 19:19–31, 1967.

[23] J.C. Fu, F. B. Bastani, and I-L. Yen. Automated discovery of loop invariants for high assurance programs synthesized using ai planning techniques. In *HASE 2008: 11th High Assurance Systems Engineering Symposium*, pages 333–342, Nanjing, China, 2008.

[24] C. A. Furia and B. Meyer. Inferring loop invariants using postconditions. In Nachum Dershowitz, editor, *Festschrift in honor of Yuri Gurevich's 70th birthday*, Lecture Notes in Computer Science. Springer-Verlag, August 2010.

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

[26] M T C Group. Demo of Aligator. Technical report, Ecole Polytechnique Federale de Lausanne, Lausanne, Switzerland, 2010.

[27] S. Gulwani, S. Jain, and E. Koskinen. Control-flow refinement and progress invariants for bound analysis. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, Dublin, Ireland, June 2009.

[28] S. Gulwani, S. Srivastava, and R. Venkatesan. Program analysis as constraint solving. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 281–292, New York, NY, USA, 2008. ACM.

[29] W. R. Harris, A. Lal, A. V. Nori, and S. K. Rajamani. Alternation for termination. In *SAS*, pages 304–319, 2010.

[30] Th. A. Henzinger, Th. Hottelier, and L. Kovacs. Valigator: Verification tool with bound and invariant generation. In *Proceedings, LPAR08: International Conferences on Logic for Programming, Artificial Intelligence and Reasoning*, Doha, Qatar, November 2008.

[31] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576 – 583, October 1969.

[32] K. Hoder, L. Kovacs, and A. Voronkov. Interpolation and symbolic elimination in vampire. In *Proceedings, IJCAR*, pages 188–195, 2010.

[33] L. Hu, M. Harman, R. Hierons, and D. Binkley. Loop squashing transformations for amorphous slicing. In *Proceedings, 11th Working Conference on Reverse Engineering*. IEEE Computer Society, 2004.

[34] R. Iosif, M. Bozga, F. Konecny, and T. Vojnar. Tool demonstration for the FLATA counter automata toolset. In *Proceedings, Workshop on Invariant Generation: WING 2010*, Edimburg, UK, July 2010.

[35] A. Jaoua, S. Elloumi, A. Hasnah, J. Jaam, and I. Nafkha. Discovering regularities in databases using canonical decomposition of binary relations. *JoRMiCS*, 1:217–234, 2004.

[36] A. Jaoua, A. Mili, N. Boudriga, and J. L. Durieux. Regularity of relations: A measure of uniformity. *Theoretical Computer Science*, 79(2):323–339, 1991.

[37] T. Jebelean and M. Giese. *Proceedings, First International Workshop on Invariant Generation*. Research Institute on Symbolic Computation, Hagenberg, Austria, 2007.

[38] L. Kovacs. Automated invariant generation by algebraic techniques for imperative program verification in theorema. Technical report, University of Linz, Linz, Austria, October 2007.

[39] 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.

[40] L. Kovacs and T. Jebelean. An algorithm for automated generation of invariants for loops with conditionals. In D. Petcu, editor, *Proceedings of the Computer-Aided Verification on Information Systems Workshop (CAVIS 2005), 7th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2005)*, pages 16–19, Department of Computer Science, West University of Timisoara, Romania, 2005.

[41] L. Kovacs and A. Voronkov. Finding loop invariants for programs over arrays using a theorem prover. In *Proceedings, 11th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, Timisoara, Romania, September 2009.

[42] L. Kovacs and A. Voronkov. Finding loop invariants for programs over arrays using a theorem prover. In *Proceedings, FASE 2009*, pages 470–485. LNCS 5503, Springer Verlag, 2009.

[43] D. Kroening, N. Sharygina, S. Tonetta, A. Letychevskyy Jr, S. Potiyenko, and T. Weigert. Loopfrog: Loop summarization for static analysis. In *Proceedings, Workshop on Invariant Generation: WING 2010*, Edimburg, UK, July 2010.

[44] D. Kroening, N. Sharygina, S. Tonetta, A. Tsitovich, and Ch. M. Wintersteiger. Loop summarization using abstract transformers. In *Automated Technology for Verification and Analysis*, volume 5311/2008 of *Lecture Notes in Computer Science*, pages 111–125, 2008.

[45] A. Louhichi, O. Mraihi, W. Ghardallou, L. Labed Jilani, Kh. Bsaies, and A. Mili. Invariant relations: An automated tool to analyze loops. In *Proceedings, Verification and Evaluation of Computer and Communications Systems*, Tunis, Tunisia, 2011.

[46] E. Maclean, A. Ireland, and G. Grov. Synthesizing functional invariants in separation logic. In *Proceedings, Workshop on Invariant Generation: WING 2010*, Edimburg, UK, July 2010.

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

[48] A. Mazurkiewicz. Proving algorithms by tail function. *Information and Control*, 18:793–798, 1971.

[49] A. Mazurkiewicz. Iteratively computable relations. *Bull. Acad. Polon. Sci, Ser. Sci. Math. Astronom. Phys.*, 20:793–798, 1972.

[50] A. Mili. Reflexive transitive loop invariants: A basis for computing loop functions. In *First International Workshop on Invariant Generation*, Hagenberg, Austria, June 2007.

[51] A. Mili, S. Aharon, and Ch. Nadkarni. Mathematics for reasoning about loop. *Science of Computer Programming*, pages 989–1020, 2009.

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

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

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

[55] O. Mraihi, W. Ghardallou, A. Louhichi, L. Labed Jilani, Kh. Bsaies, and A. Mili. Computing preconditions and postconditions of while loops. In *Proceedings, ICTAC: International Colloquium on Theoretical Aspects of Computing*, Johannesburg, SA, 2011.

[56] J. N. Oliveira. Extended static checking by calculation using the pointfree transform. In *Lecture Notes in Computer Science*, number 5520, pages 195–251. Springer Verlag, 2009.

[57] J.N. Oliveira. Pointfree foundations for (generic) lossless decomposition. Technical Report TR-HASLab:3:2011, INSEC TEC, HASLab, 2011.

[58] J.N. Oliveira and C.J. Rodrigues. Pointfree factorization of operation refinement. In *Lecture Notes in Computer Science*, number 4085, pages 236–251. Springer Verlag, 2006.

[59] A. Podelski and A. Rybalchenko. Transition invariants. In *Proceedings, 19th Annual Symposium on Logic in Computer Science*, pages 132–144, 2004.

[60] J. Riguet. Quelques proprietes des relations difonctionnelles. *Comptes Rendus Hebdomadaires des Seances de l'Academie des Sciences*, 230:1999–2000, 1950.

[61] S. Sankaranarayana, H. B. Sipma, 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.

[62] G. Schmidt. *Relational Mathematics*. Number 132 in Encyclopedia of Mathematics and its Applications. Cambridge University Press, November 2010.

[63] H. Velroyen and Ph. Rümmer. Non-termination checking for imperative programs. In Bernhard Beckert and Reiner Hähnle, editors, *Tests and Proofs, Second International Conference, TAP 2008, Prato, Italy*, volume 4966 of *lncs*, pages 154–170. spv, 2008.

[64] F. Zuleger and M. Sinn. LOOPUS: A tool for computing loop bounds for C programs. In *Proceedings, Workshop on Invariant Generation: WING 2010*, Edimburg, UK, July 2010.