Convergence: Integrating Termination and Abort-Freedom

Nafi Diallo

Monmouth University, Long Branch, NJ

Wided Ghardallou

FST, University of Tunis El Manar, Tunis, Tunisia

Jules Desharnais

Laval University, Quebec City, Canada

Ali Mili [1]

NJIT, University Heights, Newark NJ 07102-1982, USA

Abstract

The condition under which a computation terminates, and the question of whether a computation terminates for a given initial state, have been the focus of much interest since the early days of computing. In this paper we argue that it is advantageous to study the property of termination in conjunction with the property of abort-freedom, i.e. the property of a program to execute without raising exceptional conditions; also, we model the two properties in a single mathematical framework, and illustrate how this integrated framework gives a better outcome than the separate analysis of the two aspects.

Key words: Termination, Abort-freedom, Convergence, Iterative Programs, Invariant Relations.

[1] correspondence author

⋆ This research was partly supported by an NSF grant to the fourth author under reference DGE 1565478.

Email addresses: ndiallo@monmouth.edu (Nafi Diallo), wided.ghardallou@gmail.com (Wided Ghardallou), jules.desharnais@ift.ulaval.ca, http://www.ift.ulaval.ca/~desharnais/ (Jules Desharnais), mili@njit.edu, http://web.njit.edu/~mili/ (Ali Mili).

1. Introduction: The Case for Merger

The condition under which a computation terminates, and the question of whether a computation terminates for a given initial state, have been the focus of much research interest since the early days of computing. The question of termination arises, by definition, in the context of repetitive tasks, that take the form of recursion or iteration; in this paper we focus on iteration. Traditionally, researchers have analyzed iterative programs by means of two constructs: they use *invariant assertions* [39] to capture functional properties of iterative programs, and *variant functions* (also referred to as *ranking functions*) [15, 60] or *well-founded orderings* [46] to model operational properties, including termination. We argue that the derivation of a ranking function of a loop is amenable to the derivation of a transitive asymmetric superset of the function of the loop body, which Podelski and Rybalchenko introduce under the name of *transition invariant* [61]. What makes the derivation of ranking functions or, equivalently, transition invariants, very difficult is the fact that the transitive closure of a union of relations is not the union of the transitive closures of the individual relations; so that whenever the function of the loop body is structured as a union of relations, it is not sufficient to compute a transitive superset of each term of the union; this has been the driving motivation behind much of the work on the generation of composite ranking functions [7, 6, 21, 28] and composite transition invariants [42].

Non-termination is not the only issue we have to worry about with regards to the execution of a program; we also have to worry about the possibility that the program encounters an exceptional condition, such as an array reference out of bounds, a reference to a nil pointer, an arithmetic overflow or underflow, the attempt to execute an illegal arithmetic operation (such as a division by zero, the square root of a negative number, the log of a non-positive number), etc. We refer to all these events as *abort conditions*, and we refer to the property of a program that avoids them as *abort freedom.* Most authors refer to this property as *safety*, but we prefer to be compatible with the terminology of Avizienis et al. [3], where *safety* refers to correctness with respect to high stakes requirements. Traditionally, abort-freedom has been investigated separately from termination, and has, understandably, used totally distinct mathematical models, such as abstract interpretation [4, 26, 22], enhanced denotational semantics [34], enhanced axiomatic definitions [1], etc.

When a program terminates without causing an abort, we say that it *converges*; and we use the term *convergence* to refer to the property of a program that terminates without causing an abort. When a program fails to converge, we say that it *diverges.*

1.1. Motivation

One of the main contributions of this paper is the ability to capture termination and abort freedom by a single model, and to generate conditions that guarantee both properties; to discuss the motivation for this decision, we consider a while loop whose execution may lead to an abort, and discuss what it means to analyze its convergence condition in an integrated manner (as opposed to analyzing separately its termination condition, and its abort-freedom condition).

- Knowing that a loop does not exceed 100 iterations is of little help if it turns out that it will cause an abort at the 10th iteration. Hence the condition of termination is insufficient unless we also know the condition of abort-freedom.

- Knowing that a loop does not cause an abort for the next 100 iterations is not necessary if it turns out that the loop exits after only 10 iterations. Hence the condition of abort-freedom is unnecessary unless we also know the condition of termination.
- The condition of convergence of a loop is <u>*not*</u> the conjunction of the condition of termination with the condition of abort-freedom. As we will see throughout this paper, the condition of convergence weaves clauses of termination and clauses of abort-freedom in non-trivial ways.
- The convergence condition of a program characterizes those initial states for which the program associates a well-defined final state; whether a program fails to deliver a final state because it fails to terminate or because it terminates in an abort condition makes no difference; in both cases the execution is considered unsuccessful and its final state is undefined.

1.2. Illustration

We consider the following loop on integer variables i, x and y, and we wish to compute the condition under which this loop terminates without attempting a division by zero; in other words, we want the condition under which this loop terminates after a finite number of iterations, and such that no single iteration will fail to execute properly.

```
while (i!=0) {i=i+2; x=x-5; y=y-y/x;}
```

The abort condition we are concerned with in this loop is the possibility of a division by zero in the statement {`y=y-y/x;`}. Application of our analytical approach (which we discuss in this paper) to the source code of this loop yields the following condition of convergence:

$cov(i,x,y) \equiv$
$(i=0)$
$\vee(i<0 \wedge i \textbf{ mod } 2=0 \wedge (x<5 \vee (5<x<\frac{-5\times i}{2} \wedge x \textbf{ mod } 5 \neq 0) \vee x>\frac{-5\times i}{2}))$.

If we analyze this condition, we find that it stipulates that either $(i=0)$ (in which case the loop does not iterate at all) or $(i<0 \wedge i \textbf{ mod } 2=0)$ (in which case the number of iterations is finite —note that if i is odd, then it will skip over zero and never terminate) then either $x<5$ (in which case x never takes value 0 as it is decremented by 5 at each iteration) or

$(5<x<\frac{-5\times i}{2} \wedge x \textbf{ mod } 5 \neq 0)$

(in which case x flies over zero on its way down but does not hit zero) or

$(x>\frac{-5\times i}{2})$

(in which case i reaches 0 and terminates the loop before x gets near zero). If $(i>0)$ or if $(i<0 \wedge i \textbf{ mod } 2 \neq 0)$ then this loop does not terminate since i never hits 0 as it is incremented by 2 at each iteration. Note how this condition weaves concerns of termination (ensuring a finite number of iterations) with concerns of abort-freedom (ensuring that no single iteration will cause a division by zero).

If we separate termination concerns from abort-freedom concerns, we find a different analysis, yielding two distinct conditions:

- *Termination*: To ensure that this loop terminates after a finite number of iterations, we must ensure that i is less than or equal to zero, and that its absolute value is even:

$$trm(i,x,y) \equiv (i \leq 0) \wedge ((-i) \textbf{ mod } \mathbf{2}=\mathbf{0}).$$

- *Abort-Freedom*: To ensure that this loop does not attempt a division by zero, we must ensure that either x grows farther and farther away from zero with each iteration, or, if it has to change signs (go from one side of zero to another), that it does so while flying over zero:

$$af(i,x,y) \equiv ((x < 5) \vee (x > 5 \wedge (x \ \mathbf{mod}\ \mathbf{5} \neq \mathbf{0}))).$$

Clearly, the condition $cov(i,x,y)$ provided by our method is not the conjunction of the separately formulated conditions $trm(i,x,y)$ and $af(i,x,y)$. Whereas the termination condition deals exclusively with variable i (which determines termination) and whereas abort-freedom deals exclusively with variable x (which is the focus of the division by zero), the convergence condition considers the two variables jointly. This is fitting, given that variables i and x do not evolve independently: there is no point in imposing a condition on x unless the loop does execute for that value of x (variable i may cause the loop to exit before x reaches that value); and there is no point in imposing a condition on i (to ensure termination) if x causes a division by zero before the question of termination arises (the execution of the loop will have failed anyway, whether the number of remaining iterations is finite or not).

For the sake of comparison, we have attempted to deploy the *VA* (Value Analysis) plug-in of *Frama-C* [10] on this example. To the extent that we are able to use this tool, and understand its functionality, we have made the following observations:

- *Different Foci.* Whereas we focus on loops and are interested to analyse their convergence condition (termination + abort-freedom), *VA* can handle arbitrary program structures and focuses primarily on abort-freedom conditions.
- *Different Goals.* Whereas we attempt to compute the condition under which a loop converges, i.e. terminates after a finite number of iterations, where each iteration executes without causing an abort, *VA* attempts to detect the possibility of aborts for a given set of initial states. Specifically, *VA* raises a warning whenever it finds that the range of values that a variable may take includes values that may cause an abort; because these are necessary but not sufficient conditions of aborts, this approach is prone to cause false alarms.
- *Different Models for Iteration.* Whereas we approximate loops by means of invariant relations, from which we derive a superset of the loop function, *VA* approximates loops by unrolling them to a user-specified value. As a consequence of this distinction, whereas our results apply uniformly regardless of the number of iterations of the loop, *VA* may produce different conclusions for different values of loop unrollings.
- *Vastly Different Means.* Whereas we are merely offering modest results that apply to loops and require careful human intervention, *VA* is an industrial strength automated tool that applies to a wide range of C programs and can handle a wide range of data structures and control structures.

To test *VA* on the sample program given above, we provided specific initial conditions to the loop (as initial values of i, x and y). We have chosen initial conditions that span various combinations of properties (termination with abort-freedom, termination without abort-freedom, abort-freedom without termination, etc). *VA* was successful in diagnosing some of these circumstances, but not all of them.

As additional illustrations, we consider the following loops and analyze their condition of convergence:

ID	Loop	Convergence Condition

P1	`for (int j=-100;j<=100;j++) {i=j;`
	 `while (i!=0) {i=i+2; x=x-5; y=y-y/x;}}`
		False

P2	`for (int j=0;j<=100;j++) {i=j;`
	 `while (i!=0) {i=i+2; x=x-5; y=y-y/x;}}`
		False

P3	`for (int j=1;j<=100;j++) {i=j;`
	 `while (i!=0) {i=i+2; x=x-5; y=y-y/x;}}`
		False

P4	`for (int z=10;z<=100;z++) {x=z;`
	 `while (i!=0) {i=i+2; x=x-5; y=y-y/x;}}`
		$i = 0 \vee i = -2$

P5	`for (int z=-100;z<=100;z++) {x=z;`
	 `{while (i!=0) {i=i+2; x=x-5; y=y-y/x;}}`
		$i = 0$

P6	`for (int z=0;z<=100;z++) {x=z;`
	 `{while (i!=0) {i=i+2; x=x-5; y=y-y/x;}}`
		$i = 0$

Experimentation bears this analysis out, in the following sense:

- Execution of programs $P1$, $P2$, and $P3$ fails to converge for any initial value of x and of y; if the initial value of x is a positive multiple of 5, then execution of these programs fails due to an abort; if the initial value of x is negative or is not a multiple of five, then the program fails to terminate. In both cases, we simply say that it fails to converge.
- Execution of program $P4$ converges only for initial values 0 and -2 of variable i; for all other initial values of i, it fails because it attempts a division by zero (execution on a gnu compiler leads to a `Floating Exception`). For $i = 0$ it converges because the inner loop does not iterate at all; for $i = -2$ it converges because, even though the inner loop executes, it exits before x becomes 0.
- Execution of programs $P5$ and $P6$ converges only for initial value 0 of variable i. For positive initial values of i, and for negative initial values that are not divisible by 2, the programs fail to converge because they fail to terminate; for other values (also different from -2 and 0), the programs fail to converge because they attempt a division by zero.

1.3. Premises

There is a vast literature on termination analysis, and on abort-freedom analysis; we discuss some of the relevant work in section 7.2. In this section, we characterize our approach by a number of premises, which ought to elucidate how our work differs from related literature; we characterize our work by its unique ends, then by its unique means.

1.3.1. Ends

Our goal is to compute the condition under which a program is guaranteed to terminate without causing an abort; in particular, we are interested in the convergence condition of while loops. This goal is more general than proving termination, for two reasons: First because we are modeling not only termination, but in fact convergence, which also includes abort-freedom; second, because we are not proving the convergence of a program for a specific initial state, rather we are attempting to characterize all the initial states that ensure convergence.

1.3.2. Means

The approach that we take in this paper can be characterized by three premises:

- *Invariant Relations.* Whereas termination is usually analyzed by means of variant functions or transition invariants, we analyze it by means of invariant relations. Specifically, the paper presents two main theorems: The first theorem maps invariant relations to necessary conditions of convergence; the second theorem gives a general format for invariant relations that model abort freedom aspects of convergence. This theorem yields a number of corollaries that capture specific forms of abort conditions. We complement these two theorems with heuristics that we deploy in practice to ensure sufficiency of the conditions of convergence; whereas necessity of these condidtions is proven, sufficiency is merely conjectured.
- *Merging Functional and Operational Properties.* Whereas other approaches analyze the functional properties of loops by means of invariant assertions and the termination properties by means of variant functions (or similar constructs), we model all aspects of the loop by means of invariant relations. Like transition invariants [62], invariant relations are required to be transitive supersets of the function of the guarded loop body; but whereas transition invariants must be asymmetric (and well-founded) because they aim to capture termination properties, invariant relations can be reflexive and symmetric, since they aim to capture all the functional properties of while loops, including equivalence relations between inputs and outputs. Invariant relations can take a wide range of forms, hence can be used to model a wide range of properties, including termination and abort-freedom.
- *The Convergence Condition as The Domain of the Program Function.* We generate convergence conditions by mapping invariant relations to approximations of the program function, then characterizing the domain of the function. The domain of the program function includes all the states that the function maps to final states; hence it makes no distinction between states that cause infinite loops from states that cause aborts; all are excluded from the domain. This is a perfect match for our goal to capture convergence in an integrated formula. Since the domain of a function is an integral part of the function, not an orthogonal attribute, it is only fitting that we should use the same artifact, namely invariant relations, to capture the function of a loop [40] and its convergence condition.

1.4. Contributions and Limitations

The core idea of this approach is to map any invariant relation of a while loop into a necessary condition of convergence of the loop. Because invariant relations can be arbitrarily large, hence capture arbitrarily little functional information of the loop, it is only fitting that theorem 2 produces necessary, but not necessarily sufficient, conditions of convergence. We could appeal to another theorem, theorem 1, to characterize necessary and sufficient conditions of convergence, but this theorem offers little guidance in practice; hence we resort to heuristics, which we discuss in section 6.1, that enable us to compute sufficient conditions of termination using partial (but still sufficient) information about the loop. Note that because the intersection of invariant relations is an invariant relation, we do not distinguish between invariant relations that capture termination from invariant relations that capture abort-freedom; rather the set of invariant relations forms a spectrum, where the same relation can capture the two aspects to varying degrees.

The main limitation of our work is that it offers ideas and algorithms, but does not offer an integrated operational tool that we could match up against existing tools; then again, given that no tool we know of computes the condition of convergence per se, all we can do is compare our approach to tools that compute termination conditions (or prove termination) and tools that compute abort-freedom conditions (or warn of possible abort occurrences). Also note that our approach is not limited to numeric data structures, but applies generally to any application domain for which we develop means to generate invariant relations; this matter is discussed in section 4. Nevertheless, because our approach relies primarily on static analysis of the source code, rather than an analysis of its execution, it is bound to be subject to the scalability limitations that characterize such methods.

In section 2 we briefly introduce elements of relational mathematics that we use throughout this paper, and in section 3 we introduce the concept of *invariant relation*, and discuss how this concept can be used to analyze loops. In section 4 we discuss a general framework for analyzing the convergence of programs, which we then specialize to iterative programs, by means of a necessary condition of convergence. In section 5, we consider several conditions of abort avoidance and apply the necessary condition of convergence to them, then we discuss in section 6 possible extensions of our work. Finally in section 7 we summarize our findings, compare them to related work, and sketch directions of future research.

2. Mathematical Background

We assume the reader familiar with relational mathematics; the purpose of this section is merely to introduce some definitions and notations, inspired from [9].

2.1. Definitions and Notations

We consider a set S defined by the values of some program variables, say x and y; we denote elements of S by s, and we note that s has the form $s = \langle x, y \rangle$. We denote the x-component and (resp.) y-component of s by $x(s)$ and $y(s)$. For elements s and s' of S, we may use x to refer to $x(s)$ and x' to refer to $x(s')$. We refer to S as the *space* of the program and to $s \in S$ as a *state* of the program. 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 ϕ.

2.2. Operations on Relations

Because relations are sets, we apply set theoretic operations to them: union ($\cup$), intersection ($\cap$), difference ($\setminus$), and complement ($\overline{R}$). 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 $R \circ R' = \{(s, s')|\exists s'' : (s, s'') \in R \wedge (s'', s') \in R'\}$. The *nucleus* of relation R is the relation denoted by $\mu(R)$ and defined by $\mu(R) = R\widehat{R}$. The n^{th} *power* of relation R, for natural number n, is denoted by R^n and defined by $R^0 = I$, and $R^n = R \circ R^{n-1}$, for $n \geq 1$. The *transitive closure* of relation R is the relation denoted by R^+ and defined by $R^+ = \{(s, s')|\exists i > 0 : (s, s') \in R^i\}$. The *reflexive transitive closure* of relation R is the relation denoted by R^* and defined by $R^* = I \cup R^+$. We admit without proof that $R^*R^* = R^*$ and that $R^*R^+ = R^+R^* = R^+$. 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. $\{(s,s')|(s,s') \in R \wedge t(s')\}$). Given a predicate t, we denote by T the relation defined as $T = \{(s,s')|t(s)\}$. The *domain* of relation R is defined as $dom(R) = \{s|\exists s' : (s,s') \in R\}$, and the *range* of R ($rng(R)$) is the domain of $\widehat{R}$. We apply the usual conventions for operator precedence: unary operators are applied first, followed by product, then intersection, then union.

2.3. Properties of Relations.

We say that R is *deterministic* (or that it is a *function*) if and only if $\widehat{R}R \subseteq I$, and we say that R is *total* if and only if $I \subseteq R\widehat{R}$, or equivalently, $RL = L$; also, we say that R is *surjective* if and only if $LR = L$. A *vector* V is a relation that satisfies $VL = V$; in set theoretic terms, a vector on set S has the form $C \times S$, for some subset C of S; we use vectors as relational representations of sets. We note that for a relation R, RL represents the vector $\{(s,s')|s \in dom(R)\}$; we use RL as the relational representation of the domain of R. A relation R is said to be *reflexive* if and only if $I \subseteq R$, *transitive* if and only if $RR \subseteq R$ and *symmetric* if and only if $R = \widehat{R}$. We 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 R is the smallest reflexive transitive superset of R. A relation that is reflexive, symmetric and transitive is called an *equivalence relation*. The nucleus of a deterministic relation f can be written as: $\mu(f) = \{(s,s')|f(s) = f(s')\}$ and is an equivalence relation. A relation R is said to be *irreflexive* if and only if $R \cap I = \phi$, i.e. it has no pairs of the form (s,s). A relation R is said to be *inductive* if and only if there exists a vector A such that $R = \overline{A} \cup \widehat{A}$; inductive relations can be written as $R = \{(s,s')|a(s) \Rightarrow a(s')\}$ for some predicate a on S.

3. Invariant Relations

Informally, an invariant relation of a while loop of the form w : {`while (t) {b}`} is a relation that contains all (but not necessarily only) the pairs of program states that are separated by an arbitrary number of iterations of the loop. Invariant relations are introduced in [52], their relation to invariant assertions is explored in detail in [55], and their applications are explored in [32]. Before we introduce a formal definition of invariant relations, we present some definitions and notations pertaining to loop semantics.

3.1. Program Semantics

Given a program g on space S, we let the *function* of g be denoted by G and defined as the set of pairs (s,s') such that if g starts execution in state s then it converges (i.e. terminates without causing an abort) and produces state s'. From this definition it stems that $dom(G)$ is the set of states s such that if execution of g starts in state s then it converges. The *convergence condition* of program g is the predicate $s \in dom(G)$; note that we talk about the convergence condition of any program, not exclusively of iterative programs. As a convention, we represent programs by lower case letters and their function by the same letter in upper case.

We consider while loops written in some C-like programming language, and we quote the following theorem, due to [54], which we use as the semantic definition of a while loop.

Theorem 1. We consider a while statement of the form $w : \{$`while (t) {b}`$\}$. Then its function W is given by:

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

where B is the function of `b`, and T is the vector defined by: $\{(s, s')|t(s)\}$.

The main difficulty of analyzing while loops is that we cannot, in general, compute the reflexive transitive closure of $(T \cap B)$ for arbitrary values of T and B.

3.2. Definitions

If we knew how to compute reflexive transitive closures of arbitrary functions and relations, then we would apply theorem 1 to derive the function of the loop, and do away with invariant relations, invariant assertions, and all the other loop artifacts [32]; but in general we do not. The interest of invariant relations is two-fold:

- First, they enable us to compute the function of a loop in a stepwise manner, by successive approximations; this is explored in [51].
- Second, perhaps more interestingly, they enable us to answer many questions about the loop without having to compute its function; this is explored in [32]. In particular, they enable us to compute convergence conditions of while loops, which we explore in this paper.

We define invariant relations formally as follows.

Definition 1. Given a while loop of the form $w : \{$`while (t) {b}`$\}$ on space S, we say that relation R is an *invariant relation* for w if and only if it is a reflexive and transitive superset of $(T \cap B)$.

The interest of invariant relations is that they are approximations of $(T \cap B)^*$, the reflexive transitive closure of $(T \cap B)$; smaller invariant relations are better, because they represent tighter approximations of the reflexive transitive closure; the smallest invariant relation is $(T \cap B)^*$. The following proposition stems readily from the definition.

Proposition 1. Given a while loop of the form $w : \{$`while (t) {b}`$\}$ on space S, we have the following results:

(1) The relation $(T \cap B)^*$ is an invariant relation for w.
(2) If R is an invariant relation for w, then $(T \cap B)^* \subseteq R$.
(3) If R_0 and R_1 are invariant relations for w then so is $R_0 \cap R_1$.

To illustrate the concept of invariant relation, we consider the following while loop on integer variables n, f, and k:

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

We consider the following relation:

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

This relation is reflexive and transitive, since it is the nucleus of a function; to prove that it is a superset of $(T \cap B)$ we compute the intersection $R \cap (T \cap B)$ and easily find that it equals $(T \cap B)$. Other invariant relations include $R' = \{(s, s')|n' = n\}$, and $R'' = \{(s, s')|k \leq k'\}$.

3.3. Properties of Invariant Relations

In [55], Mraihi et al. discuss the relationships between invariant relations, invariant assertions [39] and invariant functions [53]. Among the most interesting properties of invariant relations, by comparison with invariant assertions, we cite the following:

- Whereas invariant assertions are unary predicates, invariant relations are binary relations.
- Whereas invariant assertions characterize the state of the program after an arbitrary number of iterations (starting from a specific initial state), invariant relations characterize pairs of states that are separated by an arbitrary number of iterations, and are independent of initial conditions.
- Whereas invariant assertions depend on the loop as well as its context (precondition, postcondition), invariant relations depend exclusively on the loop. Consider, for example, the loop above; we have generated invariant relations for it but we cannot generate invariant assertions unless we are given initial conditions. Let us consider:

  g: {`k=0; f=1; while (k!=n) {k=k+1; f=f*k;}`}.

  Then we can offer the following invariant assertion: $f = k!$.
- Given an invariant relation R and an initial condition, which we represent relationally by a vector, say C, we find that $A = \widehat{R}C$ is an invariant assertion. If we consider again the initialized loop above, where $C = \{(s,s')|k = 0 \wedge f = 1\}$ represents the initial condition and R (given above) is an invariant relation, then we find the following invariant assertion:

  A

  $=$ {proposed formula}

  $\widehat{R}C$

  $=$ {substitution}

  $\{(s,s')|\exists s' : \frac{f}{k!} = \frac{f'}{k'!} \wedge f' = 1 \wedge k' = 0\}$

  $=$ {simplification}

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

  which is indeed an invariant assertion of the loop for the given initial condition C; if we had used the initial condition $C' = \{(s,s')|f = 5 \wedge k = 4\}$ then we would have found the invariant assertion $A' = \{(s,s')|24 \times f = 5 \times k!\}$.

  In [55], Mraihi et al. prove that **all invariant assertions stem from invariant relations**, according to the formula $A = \widehat{R}C$.
- Given an invariant assertion A, we can derive an invariant relation from it by the formula: $R = \overline{A} \cup \widehat{A}$. Considering again the initialized factorial loop above and the invariant assertion $A = \{(s,s')|f = k!\}$, we generate the following invariant relation for the uninitialized loop:

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

  In [55], Mraihi et al. prove that **all inductive invariant relations stem from invariant assertions**, according to the formula $R = \overline{A} \cup \widehat{A}$.

ID	variables	constants	condition	clauses of loop body	invariant relation
1R1	i: int;		true	$i' = i + 1$	$\{(s, s') \mid i \leq i'\}$.
1R2	x: int;	a: int>1;	true	$x' = x + a$	$\{(s, s') \mid x \leq x' \wedge$ x $\mathbf{mod\ a = x'\ mod\ a}\}$
2R1	x, y: int	a, b: int	true	$x' = x + a \wedge$ $y' = y + b$	$\{(s, s') \mid ay - bx = ay' - bx'\}$
2R2	x, y: int	a: int>0;	x mod a=0	$x' = x/a \wedge$ $y' = a \times y$	$\{(s, s') \mid xy = x'y'\}$
2R3	i: int; $a[\,], b[\,]$: int;	*low*, *high*: int	$i < high$	$x' = x + a[i] \wedge$ $i' = i + 1$	$\{(s, s') \mid x + \sum_{k=i}^{high} a[k] =$ $x' + \sum_{k=i'}^{high} a'[k]$ $\wedge i \leq i' \wedge a' = a\}$

Fig. 1. Sample Recognizers

3.4. Generating Invariant Relations

In order to illustrate the practical potential of our research, we have developed a prototype tool that generates invariant relations of loops written in C-like languages (C, C++, Java). The design and operation of this tool is beyond the scope of this paper, and is discussed in other sources [40]; in this paper, we briefly present some details of this tool, for the purpose of making this paper self-contained.

Proposition 2. Let w: {`while (t) {b}`} be a while loop on space S. The relation $R = I \cup T(T \cap B)$ is an invariant relation for w.

This relation can be computed constructively from T and B, and includes pairs (s, s') such that $s' = s$ (case when no iterations are executed) and pairs (s, s') such that s verifies t and s' is in the range of $(T \cap B)$ (case when one or more iterations are executed). We refer to it as the *elementary invariant relation* of w, and in practice we generate it systematically whenever we analyze a loop. To generate other relations, we proceed by pattern matching: We map the source code of loops (in C, C++, or Java) onto relational notation, then we match clauses of their relational representation against code patterns for which we know invariant relation patterns. Whenever a match is successful, we generate an invariant relation by instantiating the corresponding invariant relation pattern with the variable substitutions of the match.

The aggregate made up of a code pattern and the corresponding invariant relation pattern is called a *recognizer.* We distinguish between 1-recognizers, whose code pattern includes a single clause of the relational representation of the loop body, 2-recognizers, whose code pattern includes two clauses, and 3-recognizers, whose code pattern includes three clauses; to keep combinatorics under control, we seldom use recognizers of more than 3 clauses. Figure 1 shows examples of recognizers.

The machinery that maps source code into internal relational notation is in place, as is the machinery that maintains the database of recognizers and matches the relational representation of a loop against recognizers to generate invariant relations. What determines the capability of our tool is the set of recognizers that are stored in its database. In the remainder of this paper, whenever we talk about an invariant relation that would fulfill some role (e.g. identify exceptional conditions that preclude convergence), it is understood that we can deploy this invariant relation in practice by including its recognizer in the database of our tool.

One may argue that our approach lacks generality because it depends on a pre-coded database of recognizers. We put forth the following observations:

- It is impossible to build a system to analyze programs without codifying the programming knowledge and the domain knowledge that are needed for this task; we argue that the recognizers are our way to capture the relevant programming knowledge and domain knowledge.
- We are currently exploring ways to do away with pre-coded recognizers for simple numeric calculations; indeed, many of our numeric invariant relations can be generated automatically from the source code by converting the code to recurrence relations (according to the work of Janicki and Carette [13]) and eliminating the recurrence variable.
- The focus of this paper is the generation of convergence conditions from invariant relations; we deploy some automated tools in the process of analyzing while loops, but these tools are not the focus of our paper; rather they are mere proofs of concept to support our conceptual research.

4. Characterizing Convergence Conditions

4.1. A Necessary Condition of Convergence

We consider a while loop w of the form w: {`while (t) {b}`} on space S, and we are interested to compute its domain, which we represent by the vector WL (where W is the function of w and L is the universal relation). The following theorem, due to [32], gives a necessary condition of convergence.

Theorem 2. We consider a while loop w of the form w: {`while (t) {b}`} on space S, and we let R be an invariant relation for w. Then

$$WL \subseteq R\overline{T}.$$

This theorem converts an invariant relation of w into a necessary condition of convergence; we seek to derive the smallest possible invariant relations, in order to approximate or achieve the necessary and sufficient condition of convergence. A proof of this theorem is given in [32]; it stems readily from theorem 1, and from relational identities. In practice, we compute the convergence condition of a loop by means of the following steps:

- Using the invariant relation generator, we generate all the invariant relations we can recognize; whenever a code pattern of the loop matches a recognizer pattern from our recognizer database, we generate the corresponding invariant relation. These relations are represented in Mathematica syntax (©Wolfram Research, `http://www.wolfram.com/`).
- We compute the intersection of the invariant relations we are able to generate, by merely taking the conjunct of their Mathematica representations.
- Given R the aggregate invariant relation computed above, we simplify the following logical formula, which is the logical representation of the term $R\overline{T}$ of theorem 2.

  $$\exists s' : (s, s') \in R \wedge \neg t(s').$$

  The result is a logical expression in s, which represents a necessary condition of convergence of the loop.

As an illustration of this theorem, we consider the sample factorial loop discussed earlier, namely:

w: {`while (k!=n) {k=k+1; f=f*k;}.`}

We consider the following invariant relation of w: $R = \{(s,s')|k \leq k'\}$. Application of theorem 2 to this invariant relation yields the following necessary condition: $k \leq n$. Indeed, this condition is necessary to ensure that the number of iterations of the loop is finite.

4.2. A Heuristic for Sufficiency

Theorem 2 gives a necessary condition of convergence, but what we are most interested in, in practice, are sufficient conditions of convergence, i.e. conditions that characterize states for which we are assured that execution converges. The only way to be sure that we have a necessary and sufficient condition of convergence is to apply theorem 1 to compute W then to compute its domain WL. But this is usually very difficult, as it requires that we compute the smallest possible invariant relation of w, which is $(T \cap B)^*$. This defeats the whole purpose of invariant relations, whose main justification is that they enable us to analyze loops cost-effectively, by generating only those invariant relations that are needed for any given purpose. Hence we propose the following heuristic.

Heuristic 1. A Heuristic for Termination. To ensure sufficiency of the termination condition produced by theorem 2 for loop {`while (t) {b}`}, proceed as follows:

- Generate the elementary invariant relation $R_0 = I \cup T(T \cap B)$, where T is the vector defined by the loop condition t and B is the function of the loop body b.
- Catalog the set of variables, say Υ, that are involved in the loop condition t; add to Υ all the variables whose values affect the value of variables in Υ.
- Derive all the invariant relations that involve the variables in Υ.
- Let R be the intersection of all the invariant relations generated above; apply theorem 2 with R.

To illustrate this heuristic, consider the following loop on integer variables i, j, and k.

```
{while (i>1) {j=j+1; i=i+2*j-1; k=k-1;}}
```

The parameters of this loop are:

- $T = \{(s,s')|i > 1\}$.
- $B = \{(s,s')|j' = j+1 \wedge i' = i+2j+1 \wedge k' = k-1\}$.

According to heuristic 1, variable i appears in the loop condition, and variable j affects the value of variable i, since j appears in the right hand side of an assignment to i; hence we focus on all the invariant relations that involve i and j. We derive the following invariant relations (using recognizers from our existing database [40]):

- The elementary invariant relation, $R_0 = I \cup T(T \cap B)$. In this example, because $(T \cap B)$ is surjective, the term $T(T \cap B)$ can simply be written as T; hence the elementary invariant relation is:
$$R_0 = \{(s,s')|s' = s \vee i > 1\}.$$
- Symmetric invariant relations:
  $R_1 = \{(s,s')|i - j^2 = i' - j'^2\}$.
- Antisymmetric invariant relations:
  $R_2 = \{(s,s')|j' \geq j\}$,

Relation R_0 captures relevant boundary conditions; relation R_2 captures the progression of the program state; relation R_1 links variable j which counts the number of iterations and variable i, which is used in the loop condition. Taking their intersection $R = R_0 \cap R_1 \cap R_2$, and applying theorem 2 to R, we find the following convergence condition:

$$(i \leq 1) \vee (i > 1 \wedge j \leq -\sqrt{i-1}).$$

This condition is provably a necessary condition of termination; because its derivation is compliant with heuristic 1, we conjecture that it is also sufficient. As an empirical test, we consider a data sample that satisfies the convergence condition, e.g. $i = 10 \wedge j = -5$ and a data sample that does not satisfy the condition, e.g. $i = 10 \wedge j = 0$, and verify that the first sample leads to termination and the second leads to an infinite loop.

4.3. Abort Freedom

Theorem 2 converts any invariant relation into an approximation of (more precisely: a superset of) the domain of the while loop. The domain of W is limited by failure of the loop to terminate, as well as failure of abort-prone statements to execute successfully; theorem 2 applies equally well to either of these circumstances. Depending on our choice of invariant relations, we can capture one aspect of non-convergence or the other, or a combination thereof. In this subsection, we present a general format of invariant relations that enable us to capture arbitrary aspects of abort-freedom (freedom from: array reference out of bounds, nil pointer reference, division by zero, arithmetic overflow, etc.).

The following discussion builds an intuitive argument for the proposed theorem, and explains how we derived it. As a general rule, a program converges whenever it is applied to a state within its domain, and fails to converge otherwise. Hence, at a macro-level, the condition of convergence of program g can merely be written as:

$$s \in dom(G).$$

If g is a sequence of two subprograms, say `g = g1; g2` then this condition can be rewritten as:

$$s \in dom(G_1) \wedge G_1(s) \in dom(G_2).$$

We can prove by induction that if g is written as a sequence of arbitrary length, say `g = (g1; g2; g3; ...; gn)`, then the condition of convergence can be written as:

$$s \in dom(G_1) \wedge G_1(s) \in dom(G_2) \wedge G_2(G_1(s)) \in dom(G_3) \wedge ... \wedge$$
$$G_{n-1}(G_{n-2}(...(G_3(G_2(G_1(s))))...)) \in dom(G_n),$$

or, equivalently, as:

$$\forall h : 0 \leq h < n : G_h(G_{h-1}(...(G_3(G_2(G_1(G_0(s)))))...)) \in dom(G_{h+1}), \qquad (1)$$

where we let G_0 be the identity relation (so that for $h = 0$ this formula provides the initial condition $s \in dom(G_1)$). If we specialize this equation to while loops, where all the G_i's are instances of the loop body, we find the following formula:

$$\forall h : 0 \leq h < n : (T \cap B)^h(s) \in dom(B). \qquad (2)$$

In practice it is difficult to compute $(T \cap B)^h$ for arbitrary values of h; fortunately, it is not necessary to compute it either, as usually only a small set of program variables (and only some of their functional properties) are involved in characterizing convergence. Hence, we substitute in the above equation the term $(T \cap B)$ by a superset thereof (which we call B'), that captures only the transformation of convergence-relevant variables. This equation can then be written as:

$$\forall h : 0 \leq h < n : B'^h(s) \in \mathit{dom}(B). \tag{3}$$

We want to change this formula from a quantification on the number of iterations to a quantification on intermediate states; to this effect, we use the change of variables: $u = (T \cap B)^h(s)$, and we represent the initial state (that corresponds to $h = 0$) by s and the final state (that corresponds to $h = n$) by s'. With this change of variables, the inequality $0 \leq h$ can be written as $(s, u) \in B'^*$, and the inequality $(h < n)$ can be written as $(u, s') \in B'^+$. Equation 3 can then be written as:

$$\forall u : (s, u) \in B'^* \wedge (u, s') \in B'^+ \Rightarrow u \in \mathit{dom}(B). \tag{4}$$

Interestingly, this equation defines an invariant relation between s and s'; this is the object of theorem 3. Before we present this theorem and its proof, we write the proposed invariant relation in algebraic form.

$$\begin{array}{ll} & R \\ = & \qquad\{ \text{ denotation } \} \\ & \{(s,s') | \forall u : (s,u) \in B'^* \wedge (u,s') \in B'^+ \Rightarrow u \in \mathit{dom}(B)\} \\ = & \qquad\{ \text{ rewriting } u \in \mathit{dom}(B) \text{ } \} \\ & \{(s,s') | \forall u : (s,u) \in B'^* \wedge (u,s') \in B'^+ \Rightarrow (u,s') \in BL\} \\ = & \qquad\{ \text{ De Morgan } \} \\ & \overline{\{(s,s') | \exists u : (s,u) \in B'^* \wedge (u,s') \in B'^+ \wedge (u,s') \notin BL\}} \\ = & \qquad\{ \text{ associativity } \} \\ & \overline{\{(s,s') | \exists u : (s,u) \in B'^* \wedge (u,s') \in (B'^+ \cap \overline{BL}\}} \\ = & \qquad\{ \text{ relational product } \} \\ & \overline{B'^*(B'^+ \cap \overline{BL})}. \end{array}$$

This discusion introduces, though it does not prove, the following theorem; its proof is given below.

Theorem 3. We consider a while loop w of the form w: {`while (t) {b}`} on space S, and we let B' be a superset of $(T \cap B)$ and we let D be a vector that is a superset of BL. If B' satisfies the following conditions:
- B'^+ is irreflexive.
- The following relation $Q = \overline{B'^*(B'^+ \cap V)}$ is transitive, for an arbitrary vector V.
- $T \cap B \cap \overline{B'^+B'} = \phi$.

then $R = \overline{(B'^*(B'^+ \cap \overline{D}))}$ is an invariant relation for w.

Vector D is an approximation of BL; we use in subsequent propositions to apply this theorem to various abort conditions; a corollary of this theorem is obtained by replacing D with BL in the proposed formula of R. This theorem provides, in effect (when applied with $D = BL$ in conjunction with theorem 2), that if the loop converges for initial state s (i.e. s is in $dom(W)$), then any intermediate state s' generated from s by an arbitrary number of iterations of the loop causes no abort at the next iteration (i.e. s' is in $dom(B)$). It is in this sense that this theorem links $dom(W)$ and $dom(B)$.

Proof. We have to show three properties of R, namely reflexivity, transitivity, and invariance (i.e. that R is a superset of $(T \cap B)$).

Reflexivity. In order to show that I is a subset of R, we show that $I \cap \overline{R} = \phi$. We find:

$I \cap \overline{R}$

$=$ { substitution }

$I \cap (B'^{*}(B'^{+} \cap \overline{D})$

$\subseteq$ { monotonicity }

$I \cap B'^{*} B'^{+}$

$=$ { relational identity }

$I \cap B'^{+}$

$=$ { irreflexivity of B'^{+} }

ϕ.

Transitivity. Transitivity is a trivial consequence of the second condition of the theorem, by taking $V = \overline{D}$.

Invariance. In order to prove that $(T \cap B) \subseteq R$, it suffices (by set theory) to prove that $(T \cap B) \cap \overline{R} = \phi$. To this effect, we analyze the expression $(T \cap B) \cap \overline{R}$. But first, we introduce a lemma to the effect that for any relation C, $C^{+}C = C^{+}C^{+}$. Indeed, $C^{+}C^{+}$ can be written $CC^{*}C^{*}C$ by decomposing C^{+} as CC^{*} then as $C^{*}C$. Now, $C^{*}C^{*}$ is equal to C^{*}: $C^{*}C^{*} \subseteq C^{*}$ because of transitivity, and $C^{*} \subseteq C^{*}C^{*}$ (because $I \subseteq C^{*}$). Hence $C^{+}C^{+} = CC^{*}C = C^{+}C$. Now, we consider the expression $(T \cap B) \cap \overline{R}$.

$(T \cap B) \cap \overline{R}$

$=$ { substitution, double complement }

$(T \cap B) \cap (B'^{*}(B'^{+} \cap \overline{D}))$

$=$ { decomposing the reflexive transitive closure }

$(T \cap B) \cap (I \cup B'^{+})(B'^{+} \cap \overline{D})$

$=$ { distributing the product over the union }

$((T \cap B) \cap B'^{+} \cap \overline{D}) \cup ((T \cap B) \cap B'^{+}(B'^{+} \cap \overline{D}))$

$=$ { relational identity: $B \cap \overline{BL} = \phi$ and hypothesis $BL \subseteq D$ }

$(T \cap B) \cap B'^{+}(B'^{+} \cap \overline{D})$

$\subseteq$ { monotonicity }

$(T \cap B) \cap B'^{+}B'^{+}$

$=$ { lemma above }

$(T \cap B) \cap B'^{+}B'$

$=$ { by hypothesis }

ϕ.

qed

The first condition of this theorem ensures that B' captures variant properties of $(T \cap B)$, hence does not revisit the same state after a number of iterations; we refer to this as the *irreflexivity condition.* The second condition ensures that the resulting relation is transitive (a necessary condition to be an invariant relation); this condition involves B' and the structure of R, but does not involve B; we refer to this as the *transitivity condition.* The third condition ensures that B', while approximating $(T \cap B)$, remains in unison with it; this condition is needed to ensure that R is a superset of $(T \cap B)$; we refer to it as the *concordance condition.* Note that there is a one-to-one correspondence between the properties of B' and the resulting properties of R: The irreflexivity of B'^{+} yields the reflexivity of R; the transitivity of $(B'^{*}(B'^{+} \cap V))$ yields the transitivity of R and the concordance of B' yields the invariance of R (i.e. the property that $(T \cap B)$ is a subset of R).

The interest of this theorem is that it captures, in the form of an invariant relation, the property of abort-freedom of a while loop (as we illustrate subsequently). To understand how it does that, consider the logical form of such invariant relations (for the case $D = BL$):

$$R = \{(s, s') | \forall u : (s, u) \in B'^{*} \wedge (u, s') \in B'^{+} \Rightarrow u \in dom(B)\},$$

where B' is a superset of B. In practice, we use B' to approximate B, by focusing on the variables that are of interest to us (that are involved in abort-prone statements) and recording how B transforms them. As for $dom(B)$, it represents the condition under which the loop body executes normally; hence it represents in particular the condition of freedom from any relevant run-time exception. If the loop manipulates arrays, this condition must provide that they are not addressed outside their bounds; if the loop computes arithmetic expressions that are prone to exceed the computer's capacity, this condition must provide that all computed values are representable; if the loop applies partial functions (that are not defined for all states), this condition must provide that all function arguments fall in the domains of the functions in question; if the loop manipulates pointers, the condition must provide that all referenced pointers are non-nil. Thus relation R, as written above, provides that all intermediate states generated by successive iterations of B cause no abort conditions. When we apply theorem 2 using invariant relations generated by theorem 3 (for various choices of B' and various possible assumptions about $dom(B)$), we find conditions on the initial states of the loop, that ensure a terminating abort-free execution.

For all its interest, theorem 3, in conjunction with theorem 2, is only offering a necessary condition of termination; we revisit the heuristic we had introduced earlier to make it applicable to convergence rather than merely to termination.

Heuristic 2. A Heuristic for Convergence. To ensure sufficiency of the termination condition produced by theorem 2 for loop {`while (t) {b}`}, proceed as follows:

- Generate the elementary invariant relation $R_0 = I \cup T(T \cap B)$, where T is the vector defined by the loop condition t and B is the function of the loop body b.
- Choose relation B' according to the criteria set forth in theorem 3 and derive the resulting invariant relation.
- Catalog the set of variables, say Υ, that are involved in the loop condition t as well as those that appear in the invariant relation derived from theorem 3; add to Υ all the variables whose values affect the value of variables in Υ.
- Derive all the invariant relations that involve the variables in Υ.

- Let R be the intersection of all the invariant relations generated above; apply theorem 2 with R.

As we have discussed in section 3, smaller invariant relations are better. If we consider the template of invariant relations generated by theorem 3, where $D = BL$,

$$R = \overline{B'^{*}(B'^{+} \cap \overline{BL})},$$

we find that R grows smaller (better) when B' grows larger (i.e. provides a looser approximation of B) and when BL (i.e. the domain of B) grows smaller (i.e. we capture more and more abort conditions). The domain of B is smallest when we capture all the possible conditions of run-time exceptions; but in practice we typically consider only some run-time exceptions at a time. For example, array references out of bounds arise only if we have arrays; nil pointer references arise only if we have pointer variables; division by zero arises only if we have expressions that involve division; arithmetic overflow arises only if we are dealing with integer variables that take potentially large values and are used in operations that are prone to overflow (addition, multiplication, power). So that in practice it is perfectly legitimate to consider some exceptional conditions and exclude others; in the next section, for the sake of illustration, we consider them one by one, in turn.

5. Computing Convergence Conditions

In this section we combine theorems 2 and 3 to capture convergence conditions that arise from different abort-prone statements. Note that nothing in theorem 2 indicates whether the necessary condition of convergence produced therein is putting a bound on the number of iterations or ensuring the absence of aborts; hence this theorem can be used with any invariant relation to enhance (i.e. make more comprehensive) our estimate of the necessary and sufficient condition of convergence. What theorem 3 does is to give some indication on how to derive invariant relations that capture abort-freedom; in practice, we use a mix of invariant relations to obtain necessary conditions of convergence, culminating (if our invariant relation is small enough) into a necessary and sufficient condition of convergence.

In the following subsections, we consider in turn a number of instances of aborts: array reference out of bounds, illegal arithmetic operation, arithmetic overflow/ underflow, and illegal pointer reference. For each instance, we specialize theorem 3 to a proposition that handles such instances specifically; these propositions are not an end in themselves as much as they are means to highlight/ illustrate the generality of theorem 3. For the sake of simplicity, we consider the abort conditions in turn, one at a time; i.e. when we analyze one of them, we assume that the others may not arise, though in fact they may, of course.

5.1. Array Reference Out of Bounds

The first corollary of theorem 3 (proposition 3, below) is so trivial as to make the theorem look like an overkill; but it also helps the reader better understand the theorem. We consider a while loop w on space S and we assume that space S includes an array a of index range $[low...high]$. We assume that space S also includes an index variable, say k, which is used to address the array. Then one issue of concern is to ensure that the array is not referenced outside of its bounds; the following proposition provides the appropriate invariant relation for this purpose.

Proposition 3. Let w be a while loop of the form w: {`while (t) {b}`} on space S, where S includes an array a of index range $[low..high]$, and an index k that is incremented by 1 at each iteration. Then the following relation is an invariant relation for w:

$$R = \{(s,s')|\forall h : k \leq h < k' \Rightarrow low \leq h \leq high\}.$$

Proof. This proposition is a special case of theorem 3, in which we take B' as $\{(s,s')|k' = k+1\}$; because array reference out of bounds is the only run-time exception under consideration, we let D be defined as: $D = \{(s,s')|low \leq k \leq high\}$. We find that the transitive closure of B' is $B'^{+} = \{(s,s')|k < k'\}$, and that the reflexive transitive closure of B' is $B'^{*} = \{(s,s')|k \leq k'\}$. We must check the three conditions of theorem 3: B'^{+} is indeed irreflexive, since its intersection with the identity is empty. To verify the transitivity condition, we consider a relation of the form $Q = \overline{B'^{*}(B'^{+} \cap V)}$ for some vector V, and we write it in logical form:

$$Q = \{(s,s')|\forall h : k \leq h < k' \Rightarrow v(h)\},$$

for some predicate v. From this representation, it is plain that Q is transitive: if predicate v holds for any h between k (inclusive) and k' (exclusive) and for any h between k' (inclusive) and k'' (exclusive) then it holds for any h between k (inclusive) and k'' (exclusive). Finally, to verify the concordance condition, we compute $T \cap B \cap B'^{+}B'$ and show it to be the empty relation:

$T \cap B \cap B'^{+}B'$

$\subseteq$ { by hypothesis }

$B' \cap B'^{+}B'$

$=$ { substitutions }

$\{(s,s')|k' = k+1\} \cap \{(s,s')|k < k'\} \circ \{(s,s')|k' = k+1\}$

$=$ { performing the relational product }

$\{(s,s')|k' = k+1\} \cap \{(s,s')|k+1 < k'\}$

$=$ { contradiction }

ϕ.

qed

It is possible (and straightforward) to derive variations of this proposition where the index variable follows a different pattern from a simple incrementation by 1. We illustrate this proposition on a simple example:

{`while (i!=0) {i=i-1; x=x+a[k]; k=k+1;}`}.

Then, we write T and B as follows:

$T = \{(s,s')|i \neq 0\}$

$B = \{(s,s')|low \leq k \leq high \wedge i' = i-1 \wedge x' = x + a[k] \wedge k' = k+1 \wedge a' = a\}$.

The elementary invariant relation R_0 of this loop is:

$I \cup T(T \cap B)$

$=$ {because T is a vector}

$I \cup TL(T \cap B)$

$=$ {associativity, and surjectivity of $(T \cap B)$}

$I \cup TL$
$=$ $\{T$ is a vector$\}$
$I \cup T$
$=$ $\{$substitution$\}$
$\{(s,s')|s' = s \vee i \neq 0\}$.
This program meets the condition of proposition 3, with $B' = \{(s,s')|k' = k+1\}$. Application of this proposition yields the following invariant relation:

$$R_1 = \{(s,s')|\forall h : k \leq h < k' \Rightarrow \mathit{low} \leq h \leq \mathit{high}\}.$$

Heuristic 2 provides that we must also compute the set Υ of variables that appear in relation R and in the loop condition, as well as any other variable whose value affects these. The loop condition involves variable i and relation R involves variable k (*low* and *high* are constants and h is a mute variable). Hence $\Upsilon = \{i,k\}$. Invariant relations that involve i and k include:
$R_2 = \{(s,s')|i' \leq i\}$.
$R_3 = \{(s,s')|i+k = i'+k'\}$.
We let R be the intersection of all the invariant relations we have generated, $R = R_0 \cap R_1 \cap R_2 \cap R_3$, and we apply theorem 2. This yields:
$R\overline{T}$
$=$ $\{$substitution, distributivity$\}$
$\{(s,s')|\exists s'' : s'' = s \wedge i'' = 0\}$
$\cup$
$\{(s,s')|\exists s'' : i \neq 0 \wedge i'' \leq i \wedge i+k = i''+k'' \wedge \forall h : k \leq h < k'' : \mathit{low} \leq h \leq \mathit{high} \wedge i'' = 0\}$
$=$ $\{$simplification (infer what we can about s)$\}$
$\{(s,s')|i = 0\}$
$\cup$
$\{(s,s')|i \geq 1 \wedge \exists s'' : i+k = k'' \wedge \forall h : k \leq h \leq k''-1 : \mathit{low} \leq h \leq \mathit{high}\}$
$=$ $\{$substitution, simplification (interval inclusion)$\}$
$\{(s,s')|i = 0\}$
$\cup$
$\{(s,s')|i \geq 1 \wedge k \leq \mathit{low} \wedge k+i-1 \leq \mathit{high}\}$
$=$ $\{$simplification$\}$
$\{(s,s')|i = 0\} \cup \{(s,s')|i \geq 1 \wedge k \leq \mathit{low} \leq k \leq \mathit{high}-i+1\}$
We obtain the following condidtion of convergence:

$$(i = 0) \vee (i \geq 1 \wedge \mathit{low} \leq k \leq \mathit{high}-i+1).$$

The reader may check that this is the condition under which we are assured that the loop is guaranteed to converge, i.e. terminate after a finite number of iterations without attempting an array reference out of bounds.

As another array example, we consider the following program on real variables x and y, array variables a and b (of type real), index (integer) variables i and j, and integer constant N, where $N \geq 1$.
```
{while (i<N) {x=x+a[i]; y=y+b[j]; j=j+i; i=i+1; j=j-i;}}.
```
We generate the elementary relation of this while loop, which takes the form $R_0 = I \cup T(T \cap B)$. In addition, because this loop has two array references, we apply theorem 3 twice, yielding the following invariant relations:

$$R_1 = \{(s,s')|\forall h : i \leq h < i' \Rightarrow \mathit{low} \leq h \leq \mathit{high}\}.$$

$$R_2 = \{(s,s')|\forall h : j' < h \leq j \Rightarrow \mathit{low} \leq h \leq \mathit{high}\}.$$

As per heuristic 2, relevant variables in this case include i and j. We generate the following invariant relations that involve i and j:

$R_3 = \{(s,s')|i \leq i'\}$.

$R_4 = \{(s,s')|i + j = i' + j'\}$.

We let R be the intersection of these five invariant relations and we apply theorem 2 to R, yielding the following condition of convergence:

$$(i \geq N)\vee$$

$$(i < N \wedge \mathit{low} \leq i \leq \mathit{high} \wedge \mathit{low} \leq j \leq \mathit{high} \wedge \mathit{low} \leq N \leq \mathit{high} \wedge \mathit{low} \leq (i+j-N) \leq \mathit{high}).$$

The first disjunct of this formula represents the case when the loop does not iterate at all (in which case it terminates readily); the second disjunct is long and complex but can in fact be interpreted easily. The first conjunct is the condition under which the loop iterates at least once; the four subsequent conjuncts impose the condition ($\mathit{low} \leq ... \leq \mathit{high}$) for the initial values and the final values of variables i and j. Because i increases monotonically and j decreases monotonically through the execution of the loop, ensuring that their initial value and final value are both within range is sufficient to ensure that all their intermediate values are also within range.

In this proposition we have assumed, for the sake of simplicity, that the loop has an index variable that is incremented at each iteration; but this is not necessary, as theorem 3 gives us much broader latitude in choosing relation B'. Any relation that satisfies the conditions of irreflexivity, transitivity, and concordance is an adequate choice for our purposes; this includes not only a relation that increments (or decrements) an integer variable by a non-zero constant amount, but any relation that otherwise *beats the tempo* of the iteration (by depleting a data structure, popping a stack, progressing through a sequence of pointers, navigating a graph, etc.).

In proposition 3 we assume that array a has a fixed range. If we consider an array whose range is variable, then we have to assume that *low* and *low* are part of the program state, and we have to apply theorem 3 using the following formula on the right hand side of the implication:

$$\mathit{low}(s'') \leq k(s'') \leq \mathit{high}(s'').$$

5.2. Illegal Arithmetic Operation

Let w be a while loop on space S of the form w: {`while (t) {b}`}, and let f be an arithmetic function that is evaluated in the loop body b; if function f involves evaluating a square root, then the value of the expression given in the argument must be non-negative; if it involves evaluating a fraction, then the value of the expression given in the denominator must be non-zero; if it involves evaluating a logarithm, then the value of the expression given in the argument must be positive; etc. We assume that execution of function $f(s)$ in state s is prone to cause an abort, and we are interested to characterize the initial states on which the loop w may execute without causing f to abort. The following proposition is a corollary of theorem 3.

Proposition 4. Let w be a while loop on space S of the form w: {`while (t) {b}`}, and let f be an arithmetic function that is evaluated in b, and let B' be a superset of B that satisfies the conditions of irreflexivity, transitivity, and concordance. Then the following relation is an invariant relation for w:

$$R = \{(s, s')|\forall s'' : (s, s'') \in B'^{*} \wedge (s'', s') \in B'^{+} \Rightarrow s'' \in \mathit{def}(f)\}.$$

where $\mathit{def}(f)$ is the set of states for which function $f(s)$ is defined (can be evaluated).

Proof. Let D be the vector defined by $\mathit{def}(f)$, i.e. $D = \{(s, s')|s \in \mathit{def}(f)\}$. We have, by definition, $BL \subseteq D$, since an element that is not in $\mathit{def}(f)$ could not be in the domain of B. This proposition is a corollary of theorem 3 for the selected D. **qed**

Note that when we talk about function f appearing in the loop body, we refer to f appearing in the relational representation of B rather than in the source code b; so that if the parameters of f are modified in b prior to the call of f, this is recorded in the definition of B. As an illustration of this proposition, we consider the following loop on integer variables i, x, and y,

```
   while (i!=0) {i=i-1; x=x+1; y=y-y/x;}
```

and we propose to apply proposition 4 using the following relation as a superset of B:

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

As a result of this choice, we find:

$$B'^{+} = \{(s, s')|x < x'\}, B'^{*} = \{(s, s')|x \leq x'\}.$$

We write the parameters of this loop (T and B) as follows:

$T = \{(s, s')|i \neq 0\}$

$B = \{(s, s')|x + 1 \neq 0 \wedge i' = i - 1 \wedge x' = x + 1 \wedge y' = y - \frac{y}{x+1}\}$.

From this definition of B, we infer that $\mathit{dom}(B)$ is defined as follows:

$$\mathit{dom}(B) = \{s|x + 1 \neq 0\}.$$

By proposition 4, we find the following invariant relation:

$$R_1 = \{(s, s')|\forall s'' : x(s) \leq x(s'') < x(s') \Rightarrow x(s'') + 1 \neq 0\},$$

which we can rewrite more simply (by the change of variable $x(s'') = h$) as:

$$R_1 = \{(s, s')|\forall h : x \leq h < x' \Rightarrow h + 1 \neq 0\}.$$

This invariant relation alone is insufficient to derive a meaningful convergence condition, since the loop terminates by testing variable i whereas this invariant relation refers to variable x; at a minimum, we need an additional invariant relation that links i and x, and an invariant relation that records the variation of i (or, equivalently, the variation of x). Finally, we also generate the elementary invariant relation, which captures the asymptotic behavior of the loop. This yields the following additional relations.

- The elementary invariant relation: $R_0 = I \cup T(T \cap B)$,
- The relation that will generate the proper condition to ensure that the number of iterations is finite: $R_2 = \{(s, s')|i \geq i'\}$,
- The relation that links the loop counter, on which termination depends, to x, on which the condition of abort-avoidance applies: $R_3 = \{(s, s')|x + i = x' + i'\}$.

We take the intersection of these four relations, and apply theorem 2; this yields the following necessary condition of convergence, which we believe to also be sufficient, by virtue of heuristic 2.

$$(i = 0) \vee (i \geq 1 \wedge (x < -i \vee x \geq 0)).$$

Indeed, in order for this loop to terminate after a finite number of iterations without attempting a division by zero, either $(i = 0)$ (in which case the loop exits without iterating) or $(i > 0)$ in which case either $(x \geq 0)$ (then $x + 1$ is initially greater than zero, and increases away from zero at each iteration) or $(x < -i)$, in which case x starts negative but the loop exits before $(x + 1)$ reaches 0.

As another example, we consider the following loop on the same space S defined by integer variables i, x and y:

```
{while (x!=0) {i=i+2; x=x-5; y=y-y/x;}},
```

we find the following convergence condition, which is sufficient in addition to being provably necessary:

$$(x = 0).$$

Indeed, any value of x other than a positive multiple of 5 leads to an unbounded number of iterations; any value of x that is a positive multiple of 5 will iterate $\frac{x}{5}$ times, but cause a division by zero on the last iteration. Hence the only case when this loop converges is the case when $(x = 0)$, i.e. it does not iterate at all.

Using the same method as the examples above, we are able to derive convergence conditions of variations of this loop (with the loop condition being `(i!=0)`), including the following configurations of indices:

```
{i=i+1; x=x+5},
{i=i-2; x=x+1},
{i=i-2; x=x+5},
{i=i+a; x=x+b}.
```

5.3. Arithmetic Overflow

Because computer arithmetic is limited, one may apply an arithmetic operation to two representable arguments, and obtain a result that is not representable in the computer; this is another source of abort conditions. In this section, we consider the condition under which the execution of a loop proceeds without causing an arithmetic overflow. Let E be an arithmetic expression that appears in b and let ϵ be the data type of the value returned by E (which we assume to be uniquely defined). Expression E may appear in a number of locations in the syntax of b: on the right hand side of an assignment statement, as a value parameter of a function call, as an argument to a write statement, as an argument to a comparison (`==`, `!=`, `>`, `<`, etc.), etc.. In order to capture freedom from overflow, we consider all the expressions of b that are prone to overflow; to each expression E, which is evaluated somewhere in b, we associate an expression E' that refers to the initial state of b rather than the state where b is invoked; hence if state s is transformed into $f(s)$ in b prior to the evaluation of E, then we let $E'(s)$ be $E(f(s))$; we let $repT(E'(s))$ be the predicate that provides that expression E' evaluated in state s produces a value that is representable; finally, we include predicate $predT(E'(s))$ in the definition of B, for each relevant expression, to characterize those initial states for which execution of b proceeds without causing an overflow. We obtain the following proposition, which is a corollary of theorem 3.

Proposition 5. Let w be a while loop on space S of the form w: {`while (t) {b}`}, and let E be an arithmetic expression of type ϵ that is evaluated in b, and let E' the expression derived from E by reference to the initial state of b (as shown above). Let B' a superset of B that satisfies the conditions of irreflexivity, transitivity, and concordance. Then

$$R = \{(s,s')|\forall s'' : (s,s'') \in B'^{*} \wedge (s'',s') \in B'^{+} \Rightarrow repT(E'(s''))\}$$

is an invariant relation for w.

Proof. This proposition is a corollary of theorem 3, where D is defined as: $D = \{(s,s')|repT(E(s))\}$. We claim that D is a subset of BL since any element s that violates $repT(E(s))$ is necessary outside the domain of B. **qed**

As an illustration of this proposition, we consider the following loop on variables x, y and z of type integer:

```
while (y!=0) {y=y-1; z=z+x;}
```

The function of the body of this loop can be written as:

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

We apply proposition 5 to this loop, taking B' as

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

To compute the transitive closure and the reflexive transitive closure of B', we refer to our database of recognizers, which provide the following relations:

$$B'^{+} = \{(s,s')|x' = x \wedge y > y' \wedge z + xy = z' + x'y'\},$$

$$B'^{*} = \{(s,s')|x' = x \wedge y \geq y' \wedge z + xy = z' + x'y'\}.$$

The condition of irreflexivity is trivially verified, since $B'^{+} \cap I$ is a subset of

$$\{(s,s')|y > y' \wedge y = y'\},$$

which is empty. Also, the condition of concordance is verified, since $(T \cap B) \cap B'^{+}B'$ is a subset of

$$\{(s,s')|y > y' + 1 \wedge y' = y - 1\},$$

which is also empty. As for the transitivity condition, its proof can be established in the same way as we did for proposition 3. Hence we may apply proposition 5, yielding the following invariant relation:

$$R = \{(s,s')|\forall s'' : y \geq y'' > y' \wedge x = x'' \wedge x'' = x' \wedge z + xy = z'' + x''y'' \wedge z'' + x''y'' = z' + x'y'$$
$$\Rightarrow repInt(z'' + x'') \wedge repInt(y'' - 1)\}.$$

We further derive the following invariant relations:

- The elementary invariant relation, that captures loop behavior in border cases:
  $R_0 = I \cup T(T \cap B)$.
- The invariant relation that records the decrease of y (to bind the number of iterations):
  $R_1 = \{(s,s')|y \geq y'\}$.
- The invariant relation that records the relation between the index variable y, and the arithmetic expression whose overflow we want to model $(z + x)$:
  $R_2 = \{(s,s')|z + xy = z' + x'y'\}$.

When we take the intersection of all these relations and apply proposition 5, we find the following convergence condition:

$$(y = 0) \vee (y > 0 \wedge \mathit{MinInt} \leq z + xy \leq \mathit{MaxInt}).$$

In addition to being provably a necessary condition of convergence, we believe that this logical formula is also a sufficient condition of convergence: In order for this loop to converge, y has to be zero, or it has to be positive, then $z + xy$ (which is the expression that the loop computes into z) has to be representable (i.e. included between *MinInt* and *MaxInt*).

As a second illustrative example, we consider the following loop on natural variables x and y:

```
while (y!=N) {y=y+1; x=x+y;}
```

The function of the loop body can be written as:

$$B = \{(s, s')|\mathit{repInt}(y + 1) \wedge \mathit{repInt}(x + y + 1) \wedge y' = y + 1 \wedge x' = x + y + 1\},$$

whence the domain of B can be written as:

$$\mathit{dom}(B) = \{s|\mathit{repInt}(y + 1) \wedge \mathit{repInt}(x + y + 1)\}.$$

For B', we choose the following relation:

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

To compute the transitive closure of this relation, we use recognizers (as the formula of B' matches a recognizer in our database), and we find:

$$B'^{*} = \{(s, s')|y \leq y' \wedge 2x - y(y + 1) = 2x' - y'(y' + 1)\},$$

$$B'^{+} = \{(s, s')|y < y' \wedge 2x - y(y + 1) = 2x' - y'(y' + 1)\}.$$

Using the same argument as in the previous example, we can establish that B' satisfies the conditions of irreflexivity, transitivity, and concordance. Hence, by proposition 5, the following relation is an invariant relation for w:

$$R_1 = \{(s, s')|(\forall s'' : y \leq y'' < y' \wedge 2x - y(y + 1) = 2x'' - y''(y'' + 1)$$

$$\wedge 2x'' - y''(y'' + 1) = 2x' - y'(y' + 1)) \Rightarrow \mathit{repInt}(y'' + 1) \wedge \mathit{repInt}(x'' + y'' + 1)\}.$$

We now consider heuristic 2: Variable y appears in the condition of the loop, and variable x appears in relation R_1 alongside y. Hence we generate the following invariant relations:

- The elementary invariant relation, $R_0 = I \cup T(T \cap B)$.
- The invariant relation that records the increase of y: $R_2 = \{(s, s')|y \leq y'\}$.
- The invariant relation that links relevant program variables (elements of Υ) to each other:
  $R_3 = \{(s, s')|2x - y(y + 1) = 2x' - y'(y' + 1)\}$.

If we take the intersection of these invariant relations and apply proposition 5, we find the following condition of convergence:

$$(y = N) \vee$$

$$(y < N \wedge \mathit{repInt}(y+1) \wedge \mathit{repInt}(x+y+1) \wedge \mathit{repInt}(N) \wedge \mathit{repInt}(x + \frac{N(N + 1)}{2} - \frac{y(y + 1)}{2})),$$

where $\mathit{repInt}(h) \equiv (\mathit{MinInt} \leq h \leq \mathit{MaxInt})$.

This condition provides that the loop terminates without causing an abort (resulting from an arithmetic overflow) if and only if $(y = N)$ (in which case the loop terminates instantly) or $(i < N)$, in which case the number of iterations is finite, but then we also have conditions that ensure that the loop causes no arithmetic overflow of (respectively) the two assignment statements of the loop body; indeed this condition mandates (respectively) that the first iteration and the last iteration are both assured not to produce an arithmetic overflow in x and y respectively.

5.4. Illegal Pointer Reference

The investigation of illegal pointer references in the general case, under general assumptions about the structure of the data, the heap management policies, data sharing, aliasing, etc. is very complex, and is beyond the scope of this paper. The only goal of this section is to show that the generic model introduced by theorem 3 applies to pointer-caused aborts as well as it applies to other abort conditions. We consider a while loop w on space S and we assume that space S includes a pointer variable p. We assume that pointer p refers to a record structure that has several pointer fields that point to records of the same type. Whenever a pointer is referenced, we must ensure that it is not nil, to avoid an abort. The following proposition is a corollary of theorem 3, and applies to loops that are prone to cause an illegal pointer reference.

Proposition 6. Let w be a while loop on space S of the form w: {`while (t) {b}` }, and let p be a pointer variable in S and f be a pointer type field in the record type referenced by p. If the function B of the loop body b is a subset of $\{(s,s')|p' = *p.f\}$, and if the data structure (graph) defined by the pointer references does not have loops (p points to itself) nor cycles (p is reachable from p) then the following relation is an invariant relation for w:

$$R = \{(s,s')|\forall p'' : reach(p,p'') \wedge reach(p'',p') \wedge p'' \neq p' \Rightarrow p'' \neq nil\},$$

where $reach(p,p')$ means that pointer p' can be reached from pointer p by an arbitrary number (possibly zero) of pointer references through field `f`.

Proof. In order to apply theorem 3, we must choose a superset B' of B, which satisfies the conditions of irreflexivity, transitivity, and concordance; we know by hypothesis that B is a subset of $\{(s,s')|p' = *p.f\}$. Hence we take:

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

The irreflexivity and concordance of B' can be established by virtue of the absence of cycles and the absence of loops in the data structure (by hypothesis). The condition of transitivity can be established in the same way as the proof of proposition 3. As for vector D, we take $D = \{(s,s')|p \neq nil\}$; since B is a subset of $\{(s,s')|p' = *p.f\}$, we can infer that BL is a subset of D. To this effect, we compute B'^{+} and B'^{*}, which we find to be as follows:

$$B'^{+} = \{(s,s')|reach(p,p') \wedge p \neq p'\},$$
$$B'^{*} = \{(s,s')|reach(p,p')\}.$$

Combining our definition of D with B'^{+} and B'^{*} as indicated by theorem 3, we find:

$$R = \overline{B'^{*}(B'^{+} \cap \overline{D})}.$$

This is exactly the invariant relation proposed by the proposition. **qed**

Heuristic 2 mandates that we supplement the invariant relation provided by theorem 3 by additional invariant relations that cover the variables that appear in the loop condition, those that pertain to the abort condition, as well as all the variables whose value affects those. To this effect, we introduce relevant invariant relations that pertain to data structures defined by pointers, which in turn require that we introduce some functions. As we recall, we assume that the data structure has no cycle; we let *Roots* be the set of nodes that have no pointer pointing to them, and *Leaves* be the set of nodes that are pointing to no other nodes (all their pointer fields are nil). We introduce a fictitious node that has links to all the roots, which we denote by *Root*, and a fictitious node to which all leaves point, which we denote by *Leaf*.

- Given a node represented by its pointer p, we let *MaxDepth(p)* be the length of the longest path from the *Root* to p, and *MinDepth(p)* be the length of the shortest path from the *Root* to p.
- Given a node represented by its pointer p, we let *maxHeight(p)* be the length of a longest path from p to the *Leaf*, and we let *minHeight* be the length of a shortest path from p to the *Leaf*.

These functions enable us to derive some general invariant relations, which we present in the following proposition.

Proposition 7. Let w be a while loop on space S of the form w: {`while (t) {b}`}, and let p be a pointer variable that is referenced in b. We assume that the record that p points to has several pointer fields, say f_1, f_2, ..., f_n. If the function of b is a subset of $B' = \{(s,s')|\exists i : 1 \leq i \leq n : p' = *p.f_i\}$ then the following relations are invariant relations for w:

$$R_1 = \{(s,s')|maxDepth(p) \leq maxDepth(p')\}.$$
$$R_2 = \{(s,s')|minDepth(p) \leq minDepth(p')\}.$$
$$R_3 = \{(s,s')|maxHeight(p) \geq maxHeight(p')\}.$$
$$R_4 = \{(s,s')|minHeight(p) \geq minHeight(p')\}.$$

Proof. Reflexivity and transitivity stem readily from the structure of the relations; invariance can be proved readily by considering that the inequalities that characterize each relation are logical conclusions of the formula: $p' = *p.f$ for any pointer field f. **qed**

In tree-like structures, where there is a single path from the root to every node, functions *minDepth* and *maxDepth* are identical, and are denoted by *depth*, affording us smaller invariant relations, as shown below.

Proposition 8. Let w be a while loop on space S of the form w: {`while (t) {b}`}, and let p be a pointer variable that is referenced in b. We assume that the record that p points to has several pointer fields, say f_1, f_2, .. f_n, and that the resulting data structure is tree-like. If the function of b is a subset of $B' = \{(s,s')|\exists i : 1 \leq i \leq n : p' = *p.f_i\}$ for some pointer field f_i of p then the following relations are invariant relations for w:

$R_1 = \{(s,s')|depth(p) \leq depth(p')\}$.
$R_2 = \{(s,s')|depth(p) + maxHeight(p) \geq depth(p') + maxHeight(p')\}$.
$R_3 = \{(s,s')|depth(p) + minHeight(p) \leq depth(p') + minHeight(p')\}$.
$R_4 = \{(s,s')|\forall h : minHeight(p) \geq h > minHeight(p')$

$$\Rightarrow \forall i_1, i_2, \ldots, i_h : (*p.f_{i_1}.f_{i_2}.\cdots.f_{i_h} \neq nil)\}.$$

Proof. Relation R_1 is reflexive and transitive; it is a superset of B' (hence a superset of B) because the unique path from the root to p' necessarily goes through p. Relation R_2 is reflexive and transitive. As for being a superset of B, it suffices to prove that it is a superset of B'. Let (s, s') be a pair of B'. Then, by definition, $depth(p') = depth(p) + 1$. Now, if p' is on the path from p to the farthest leaf, then $maxHeight(p) = 1 + maxHeight(p')$. Whence $depth(p) + maxHeight(p) = depth(p') + maxHeight(p')$. If p' is not on the path from p to the farthest leaf, then $maxDepth(p) > 1 + maxHeight(p')$. Whence $depth(p) + maxHeight(p) > depth(p') + maxHeight(p')$. The same argument can be used (with some duality) for relation R_3. As for the property that relation R_4 is an invariant relation, it is a corollary of theorem 3. **qed**

A trivial corollary of this proposition is that if p has a single pointer field, then there is a single path from any node to a leaf, hence $maxHeight()$ is the same as $minHeight()$; we refer to this function as $height()$, and we have the following proposition.

Proposition 9. Let w be a while loop on space S of the form w: {`while (t) {b}`}, and let p be a pointer variable that is referenced in b. We assume that the record that p points to has a single pointer field (say, f) and that it defines a structure without cycles. If the function $(T \cap B)$ is a subset of $B' = \{(s, s')|p' = *p.f\}$ then the following relation is an invariant relation for w:

$$R = \{(s, s')|depth(p) + height(p) = depth(p') + height(p')\}.$$

If an integer variable is incremented or decremented alongside a pointer reference in a rooted tree structure, then we can link the depth of a node to the integer variable, as shown in the following proposition.

Proposition 10. Let w be a while loop on space S of the form w: {`while (t) {b}`}, and let p be a pointer variable in S and i be an integer variable in S. If the function $(T \cap B)$ is a subset of $B' = \{(s, s')|i' = i + c \wedge p' = *p.f\}$ for some constant c, then the following relation is an invariant relation for w:

$$R = \{(s, s')|i - c \times depth(p) = i' - c \times depth(p')\}.$$

Proof. This relation is reflexive and transitive, as it is the nucleus of a function. That it is a superset of B' can be readily established by considering that if $p' = *p.f$ then $depth(p') = depth(p) + 1$. **qed**

We consider a number of examples to illustrate the results of this subsection. If we consider the following program on pointer p, where the record of p has a single pointer field *next*,

```
while (p!=nil) {p=*p.next;},
```

then we find the convergence condition true. Indeed, if we let $B' = \{(s, s')|p' = *p.next\}$, then we find:

$B'^* = \{(s, s')|reach(p, p')\}$,

$B'^+ = \{(s, s')|reach(p, p') \wedge p \neq p'\}$.

Also, we find:

$BL = \{(s, s')|p \neq nil\}$.
Application of proposition 6 yields the following invariant relation:
$R = \{(s, s')|\forall p'' : reach(p, p'') \wedge reach(p'', p') \wedge p'' \neq p' \Rightarrow p'' \neq nil\}$.
The formula proposed in theorem 2 is:
$R\overline{T}$
$=$ {substitution}
$\{(s, s')|\exists s' : \forall p'' : reach(p, p'') \wedge reach(p'', p') \wedge p'' \neq p' \Rightarrow p'' \neq nil \wedge p' = nil\}$.
$=$ {substitution}
$\{(s, s')|\exists s' : (\forall p'' : reach(p, p'') \wedge reach(p'', p') \wedge p'' \neq nil \Rightarrow p'' \neq nil) \wedge p' = nil\}$.
$=$ {tautology}
$\{(s, s')|\exists s' : p' = nil\}$.
$=$ {tautology}
L.
This loop converges for any initial state.

If we consider the following program on integer variable i and pointer variable p, where the record of p has a single pointer field *next*,
`while (i<N) {p=*p.next; i=i+1;}`,
then we find the following convergence condition

$$(i \geq N) \vee (i < N \wedge minHeight(p) \geq N - i).$$

If we consider the following program on integer variable i and pointer variable p, where the record of p has two pointer fields *left* and *right*,
`while (i<N) {i=i+1; if ((i% 2)==0){p=*p.right;} else {p=*p.left;}}`,
then we find the following formula for B:
$B = \{(s, s')|i < N \wedge i' = i + 1 \wedge (i \textbf{ mod 2} = \mathbf{0}) \wedge \mathbf{p}' = *\mathbf{p}.\mathbf{right}\}$
$\cup\{(s, s')|i < N \wedge i' = i + 1 \wedge (i \textbf{ mod 2} = \mathbf{1}) \wedge \mathbf{p}' = *\mathbf{p}.\mathbf{left}\}$
{deleting conjuncts}
$\subseteq \{(s, s')|p' = *p.right\} \cup \{(s, s')|p' = *p.left\}$.
This is precisely the formula of relation B' put forth in proposition 6. Using this proposition, we find the following convergence condition

$$(i \geq N) \vee (i < N \wedge N - i \leq maxHeight(p)).$$

Note that this is a necessary condition of convergence, but not a sufficient condition of convergence; we conjecture that a sufficient condition of convergence would have *minHeight()* rather than *maxHeight()*.

6. Extensions

6.1. Condition of Sufficiency

Throughout this paper we have considered several examples of programs for which we have given a necessary condition of convergence, and claimed that we thought the condition was sufficient, in addition to being provably necessary. In this section, we discuss two questions, namely: why can't we derive a provably sufficient condition of convergence? How can we claim that our necessary conditions are sufficient? We address these questions in turn, below.

- *Why can't we derive a sufficient condition?* It is hardly surprising that arbitrary (arbitrarily large) invariant relations can only generate necessary conditions, since they capture arbitrarily partial information about the loop, hence cannot be used to make claims about a global property of the loop. Yet strictly speaking, we can formulate a sufficient condition of convergence, but it is of little use in practice. A sufficient condition of convergence would read as follows: Given a while loop of the form w: {`while (t) {b}`}, and given the invariant relation $R = (T \cap B)^*$, then $R\overline{T} \subseteq WL$.

  As we recall from proposition 1, $R = (T \cap B)^*$ is an invariant relation of the loop, and is in fact the smallest invariant relation of the loop. In practice, it is very difficult to compute this reflexive transitive closure for arbitrary T and B. One of the main interests of invariant relations is in fact that:
  · First they enable us to compute or approximate the reflexive transitive closure of $(T \cap B)$.
  · Second and perhaps most importantly, they enable us to dispense with the need to compute the reflexive transitive closure of $(T \cap B)$; in particular, one of the main motivations for using invariant relations is that they enable us, with relatively little scrutiny of the loop, to answer many questions pertaining to the loops.

  Hence requiring that we compute the strongest possible invariant relation to secure a sufficient condition of convergence defeats the purpose of using invariant relations, and makes the approach impractical.
- *How can we claim sufficiency?.* Heuristics 1 and 2 are derived in such a way as to mandate the derivation of a sufficiently small invariant relation to ensure sufficiency of the convergence condition produced by theorem 2. As far as ensuring that the number of iterations is finite, Heuristic 1 mandates that we identify the variables that intervene in the loop condition, and generate all the invariant relations that involve these variables, and any variable that affects their value (through assignment statements). As for ensuring freedom from aborts, Heuristic 2 mandates to include any invariant relation that links the variables identified above with the variables that are involved in the abort condition (array indices, denominators of fractions, arithmetic expressions, etc). Another heuristic that we are considering is to define a set of recognizers that specialize in computing a sufficient condition of convergence, by focusing on convergence-related details; for example, if the loop body includes a clause of the form $x' = x + a[i]$ for some real variable x, real array a, and index (integer) variable i, then the complete recognizer would generate the invariant relation $\{(s, s')|x + \Sigma a = x' + \Sigma a'\}$ whereas the convergence-related recognizer would merely record that array a has been accessed at index i. A final heuristic, invoked in [40] for the purpose of minimizing the number of invariant relations generated by our tool, involves generating just enough invariant relations to link all the statements of the loop body into a connected graph.

  All the heuristics discussed herein are intended to enable us to claim sufficiency of our convergence condition without having to generate all the invariant relations of the loop; we envision to organize these heuristics into a cohesive algorithm, as part of our future research plans.

6.2. Scope and Scalability

Even though we use an automated prototype to support our experimentation, we do not view the prototype as a significant contribution of this paper; rather, we view it as a proof of concept and as an experimental tool to highlight the method that we propose. In this section, we briefly discuss the scope of our work as well as scalability issues.

- The machinery that generates and analyzes invariant relations is operational; what determines the scope of applicability of the proposed approach is the size of the database of recognizers. Whereas currently we have a database of about 89 recognizers, we can easily, in the context of a tool development effort, augment this database, or, more judiciously, create several versions, catering to different application domains.
- The approach we advocate in this paper is not limited to programs that handle numeric data; rather it can be applied to any data pertaining to any application domain, provided we have the appropriate recognizers, as well as the relevant domain abstractions. In fact for numeric programs we could conceivably do without recognizers, to the extent that we can generate invariant relations on the fly, by eliminating recurrence variables as we discuss in section 3.4.
- While the generation of invariant relations that capture relationships between evolving program variables is straightforward (whether it be through recognizers or by recurrence formulas), the automatic generation of invariant relations according to the format provided by theorem 3 is difficult. This requires not only that we derive the right formula for B', but also that we verify the required properties; we are considering to design recognizer-like artifacts to support this task.
- Because it proceeds by static analysis of the source code, because it aims to derive general claims about program behavior (no assumptions about the number of iterations of loops), and because it is based on specific program structures (while loops), this approach does not readily scale to programs of arbitrary scale and complexity. However, because it is goal-driven, in the sense that it generates invariant relations on demand, depending on the verification goal, this approach may be applied even when a limited number of recognizers can be deployed (we do not need to know everything about a program to draw conclusions about it).

6.3. Nested Loops

In this section we briefly review how to analyze nested while loops: Let w be a loop of the form:

```
w: {while (t) {...  ...  ...  w':  while (t') {b';}; ... ... ...}}
```

where w and w' are labels in the source code (to identify the loops). To analyze this nested loop, we first consider the inner loop and derive its convergence condition, which we call $C'(s)$. Then we apply theorem 3 to the outer loop, using $C'(s)$ for $s \in dom(B)$, assuming no other source of abort exist in the loop body of w (if other sources did exist, we just take their conjunct with $C'(s)$). The rationale for this process is very straightforward: when we apply theorem 3 to a loop, we capture in $dom(B)$ the condition under which the loop body is assured to converge; in the case of a nested loop, that condition is precisely $C'(s)$ (if no other cause of divergence existed). As an illustration of this approach, consider again the example of programs P1 to P6 presented in section 1.2: Given that the condition of convergence of the inner loop was found to be

$$C'(i, x, y) \Leftrightarrow$$

$$(i = 0) \vee (i < 0 \wedge i \textbf{ mod } 2 = 0 \wedge (x < 5 \vee (5 < x < \frac{-5 \times i}{2} \wedge x \textbf{ mod } 5 \neq 0) \vee x > \frac{-5 \times i}{2})),$$

the condition of convergence of P4 (for example) stems from simplifying the expression

$$\forall x, 10 \leq x \leq 100 : C'(i, x, y).$$

The result is $i = -2 \vee i = 0$.

7. Conclusion

7.1. Summary

7.1.1. Conceptual Results

In this paper, we have introduced an approach to computing the convergence condition of programs, which can be characterized by the following premises:

- Our definition of convergence refers to the property that the number of steps in the program's execution is finite, and that each individual step can be completed without causing an illegal operation (to which we refer as an abort).
- Focusing on while loops, we invoke a theorem (theorem 2, due to [32]) that transforms each invariant relation into a necessary condition of termination. Because the theorem makes no reference to what may preclude normal termination, it can be used to model any possible cause of divergence.
- We find that when we apply theorem 2 with invariant relations that are antisymmetric (in addition to being reflexive and transitive), we generate bounds on the number of iterations, i.e. conditions to the effect that the number of iterations is finite.
- We present a theorem (theorem 3) that provides a general format for invariant relations that capture the property of abort freedom; when theorem 2 is applied using invariant relations generated by theorem 3 combined with invariant relations that capture other relevant functional information, they produce a necessary condition of convergence that encompasses bounds on the number of iterations as well as conditions that ensure that no single iteration causes an abort at run-time.
- We have generated a number of corollaries for theorem 3, which apply its general formula to special abort conditions, generating appropriate convergence conditions for each type of abort. The adequacy of the results that we have obtained from the corollaries give us further confidence in the generality of the original theorem.
- We argue that while invariant relations enable us to compute provably necessary conditions of convergence, we can use them to also attain sufficient conditions of convergence, provided we generate a sufficient number of them; we propose heuristics to this effect.

7.1.2. Automated Support

The transformation of source code (C++ for now) into our internal relational notation is a simple compiler transformation. The step that generates invariant relations from the relational representation of the loop, by matching clauses of the relational representation against recognizers is currently operational; and our current database includes 89 recognizers, covering a number of data structures (scalar data types, structured data types such as arrays, abstract data types such as lists, etc). Whereas this component currently operates by syntactic matching, we are replacing it with a semantic matching algorithm. Semantic matching determines whether a formal pattern matches an actual pattern by instantiating the formal pattern with actual variable names and checking the validity of the theorem that results from the equality of the two patterns. Semantic matching offers two significant advantages over syntactic matching: first it enables us to match patterns across a wide range of variance in form; second, it enables us to achieve much broader scope with fewer recognizers. The main obstacle to the automation of our tool is the generation of invariant relations according to the conditions of theorem 3, as well as the application of the heuristics that we discuss in section 4; this step is currently carried out by hand. The third step, of transforming invariant relations into necessary conditions of convergence, is a trivial step, since it involves submitting a precoded formula, in which a placeholder is replaced by the current invariant relation, to an algebraic system, to have the formula simplified.

7.2. Related Work

7.2.1. Loop Termination

Analysis of termination is a very active research area for which there is a vast bibliography; it is impossible to do justice to all the relevant work in this area, so we will just discuss some work that has influenced our research.

Boyer and Moore [5] propose a technique based on semi-automatic theorem proving where termination arguments have to be user-supplied. When we propose to compute the domain of W, the function of a loop w, we are in effect computing the weakest precondition of the loop for the postcondition **true** [27, 33]. Whereas Dijkstra and Gries use invariant assertions to capture the semantics of the loop, we use invariant relations; and whereas Gries uses well founded orderings and related artifacts to prove the termination of the loop, we use invariant relations, albeit with a slightly distinct profile (reflexive, transitive and antisymmetric for termination, vs. reflexive, transitivee and symmetric for functional properties). The work of Gupta et al. [37] uses templates to identify recurrent sets, but for the sole purpose of characterizing infinite loops; also focused on non termination is the work of Velroyen and Ruemmer [69]. In these two cases, the analysis is restricted to linear programs. Linear programs are also the focus of other researchers, such as [29, 8, 67, 15]. In [12], Burnim et al. propose a dynamic approach to detecting infinite loops, based on concolic executions (a combination of concrete execution and symbolic analysis); the technique is generally incomplete, in the sense that the iterative analysis may lack the resources needed to solve complex constraints. In [30] Falke et al. critique existing approaches to the analysis of termination of iterative program, on the grounds that treating bitvectors and bitvector arithmetic as integers and integer arithmetic is unsound and incomplete; also, they propose a novel method for modeling the wrap-around behavior of bitvector arithmetic, and analyze loop termination within this model.

In [60], Podelski and Rybalchenko propose a complete method for computing linear ranking functions; their approach is complete in the sense that if the loop can be bound by a linear ranking function, one such a function will be found by their method; Lee et al. [44] use the results of Podelski and Rybalchenko [60, 19] and propose an approach based on algorithmic learning of Boolean formula in order to compute disjunctive, well-founded, transition invariants; the technique appears to be particularly effective when dealing with simple programs dealing with linear arithmetic. In [20], Cook et al. give a comprehensive survey of loop termination, in which they discuss transition invariants; whereas invariant relations are approximations of $(T \cap B)^*$, transition invariants are in fact approximations of $(T \cap B)^+$; this slight difference of form has a significant impact on the properties and uses of these distinct concepts. Whereas transition invariants are used by Cook et al. to characterize the well-founded property of $(T \cap B)^+$, we use invariant relations to approximate the function of a loop, and its domain.

In [14], Chawdhary et al. use abstract interpretation to synthesize ranking functions; their technique is subsequently improved by Tsitovitch et al. [68], where loop summaries allow them to increase the scalability of the technique. In [16], Cook et al. propose to underapproximate weakest liberal preconditions in order to synthesize simpler predicates that still enable them to prove termination in cases where other tools would return a spurrious warning of possible non-termination. In [69], Velroyen and Ruemmer propose to synthesize invariants from a set of prerecorded invariant templates, and deploy a theorem prover to prove that the final states characterized by the invariants is unreachable, hence disproving termination; because it provides a necessary condition of termination, our work can be used to disprove termination: whenever the necessary condition is violated, the loop does not terminate. In [18] Cook et al. introduce a technique for proving the non-termination of non-linear, non-deterministic and heap-based programs. Their approach is based on an over-approximation of non-linear behaviors by means of non-deterministic behaviors, and is based on the concept of closed recurrence set. We are interested in this approach because of its analogy with our work: an invariant relation is an overapproximation of the program's function, and theorem 2 maps each invariant relation into a necessary condition of convergence, whose negation is a sufficient condition of divergence.

In [2], Ancourt et al. analyze loops by some form of abstract interpretation, but they dispense with the fixpoint semantics of loops by attempting to approximate the transitive closure of the loop body abstraction. While the calculation of transitive closures is complex in general, the authors attempt it using affine approximations of the loop body transformations, which they define in terms of affine equalities and inequalities of state variables. Using techniques of discrete differentiation and integration, they derive an algorithm that computes affine invariant assertions from this analysis, and use the generated assertions to monitor abort-freedom conditions on the state of the program. They illustrate their algorithm by running it on many published sample loops. Overall, it is fair to say, perhaps, that all the work on ensuring termination by means of ranking functions and well-founded orderings is an attempt to approximate (i.e. find a superset of) the transitive closure of the loop body, i.e. $(T \cap B)^+$.

In summary, we can characterize our approach (and contrast it with other approaches) by means of the following premises: unlike all other approaches, we compute an integrated convergence condition rather than merely a termination condition; we use the same artifact, namely invariant relations, to capture functional properties and operational properties (termination, abort-freedom) of iterative programs; we can handle any data type (not limited to numeric types) and any numeric transformation (not limited to linear transformations). Limitations of our approach include: we can only handle programs for which we have prestored recognizers; and we can only ensure that the conditions we generate are necessary conditions of convergence. Our future work aims to address these weaknesses.

7.2.2. Abort-Freedom

Gries and Schneider [34] and Almeida et al. [1] recognize the importance of modeling abort-freedom, and integrate this consideration in their proof systems. Gries and Schneider [34] alter the assignment rule to add a precondition to the effect that the initial state falls within the domain where the expression can be evaluated. Almeida et al. [1] model run-time errors (what we call aborts) by adding a fictitious *error* state to their state space, and by redefining the semantics of their language to take into account cases where run-time errors arise; also, they redefine a Hoare-like inference system for their augmented model.

In [45] Luecke et al. evaluate commercial and non-commercial systems for detecting run-time errors in C/C++ programs, including: uninitialized variables, overflows, underflows, division by zero, parameter passing mismatches, out-of-bounds array references for static and dynamically allocated arrays, nil pointer references, memory allocation errors, memory leaks, and file descriptor errors. They find that Insure++ (from Parasoft) and Purify (from IBM Rational), two commercial systems, are the best tools for the task, and that Mpatrol is the best non-commercial tool they have tested. Also, they find that Insure++ and Purify are the best tools of their whole pool, including commercial and non-commercial tools. Purify operates by monitoring every byte of memory, keeping track of allocation, initialization, assignments, usage, deallocation, etc. and ensuring that all operation sequences follow normal usage patterns. Insure++ operates by instrumenting the program with code that checks for anomalies, especially dealing with memory usage and memory allocation.

In [70] Wissing discusses the design and operation of *PolySpace*, an automated tool for the analysis of run-time behavior of C, C++ and Ada. This tool operates by cataloguing the set of erroneous states defined by the programming language (E), and matching them against an approximation of the set of states (P) that the program may be in (inferred from static analysis). By comparing E and P, PolySpace can formulate a diagnosis about the possibility, certainty or impossibility of a run-time error. Run time errors that the tool considers include division by zero, overflows, pointer dereferences and array indices out of bound.

Frama-C (http://frama-c.com/) is a suite of tools dedicated to the static analysis of source code written in C. It can be used to certify the correctness of the code with respect to functional specifications written in ACSL; and it can also be used to analyze the run-time behavior of programs with respect to common abort conditions. It proceeds by making conservative estimates of possible execution states, then checking that such sets of states do not cause abort conditions (and issuing appropriate warnings otherwise). Upon parsing the code, Frama-C delivers a sequence of analysis results, including: *Value Analysis*, where it records the set of values that each variable may take at each step of the program; *Effects Analysis*, where Frama-C provides for each statement an exhaustive list of the memory cells that may be modified by this statement during execution; *Dependency Analysis*, where Frama-C records all the statements that define the value of each variable at each point in the program; and *Impact Analysis*, where each statement is associated with the statements that it affects by its execution. Frama-C uses this anslysis to issue alerts about possible run-time errors. In [11] Burghardt and Pohl introduce VeriFast, a verification tool for Java, and discuss its similarities and differences with Frama-C (other than the difference of programming language) and ACSL.

Abstract interpretation [23, 25, 24] is a broad scoped technique that aims to infer properties of programs by successive approximations of their execution traces.Also, abstract interpretation has been used to, among others, analyze the properties of abort freedom of arbitrary programs [41]. The work on abstract interpretation has given rise to a widely used automated tool, *Astrée*, that analyzes programs and issues reports pertaining to their correctness, termination, abort-freedom, etc [26, 4]. Astrée considers different classes of run-time errors, and handles them differently: For errors terminating the execution, it warns the user and continues by considering only those executions that did not trigger the error; for errors not terminating the execution with predictable outcome, it warns the user and continues with worst-case assumptions; for errors not terminating the execution with unpredictable outcome, it warns the user and continues by considering only those executions that did not trigger the error. Even in the absence of functional specifications, Astrée enforces implicit requirements, including: compliance to the norms of C; avoiding implementation-specific undefined behaviors; adherence to programming guidelines; and enforcing programmer assertions.

7.2.3. Pointer Semantics

Heap data structures manipulate potentially unbounded data structures, which do not lend themselves to simple modeling; as such, they represent one of the biggest challenges to scalable and precise software verification. In order to model the property that a loop causes no illegal pointer reference, we have to capture some aspects of pointer semantics; in our work, we use invariant relations to represent unbounded pointer references, and to reason about them. In this section, we review some of the alternative approaches to pointer semantics, and compare them to ours; we have been able to classify them into five broad categories, which we review in turn below.

- *Shape Analysis.* These approaches proceed by identifying some structure into the pattern of pointers between nodes. In [64] Sagiv et al. use three-valued logic as a foundation for a parameterized framework for carrying out shape analysis; the framework is instantiated by supplying predicates that capture different relationships between nodes, and by supplying the functions that specify how the predicates are updated by particular assignments. In [35], Bhargav et al. propose a new shape analysis algorithm, which is presented as an inference system for computing Hoare triples summarizing heap manipulation programs. These inference rules are used as a basis for a bottom-up shape analysis of data structures.
- *Path-Length Analysis.* In [65], Spoto et al. prove the termination of programs written in Java Bytecode by mapping them into a constraint logic program which is built on the basis of a path-length analysis of the original program. The proof is based on the proposition that the termination of the logic constraint program is a sufficient condition to the termination of the original program. The path-length analysis of a Java bytecode program derives an upper bound of the maximal length of a path of pointers that can be followed from each variable of the program; the concepts of maxDepth and maxHeight presented in this paper bear some resemblance to Spoto et al.'s path-length function, and the overapproximations derived for the path-length function bears some resemblance to the type of approximations that are produced by invariant relations. But while Spoto et al. are interested in proving program termination, we are interested in computing termination conditions; while Spoto et al. are interested in termination as the property that the program executes a finite number of steps, we are interested to model termination as well as abort freedom; while Spoto et al.'s approach is focused primarily on the data structure of the program, our approach is focused primarily on its control structure.
- *Alias Analysis.* This approach focuses on determining whether two pointers refer to the same heap cell [56]. In [38], Hackett and Aiken use a combination of predicate abstraction, bounded model checking, and procedure summarization to compute a precise path-sensitive and context-sensitive pointer analysis. Alias analysis is only useful for reasoning about explicitly named heap cells, and cannot model general unbounded data structures.

- *Separation Logic.* This approach makes it possible to reason about heap manipulation programs [63] by extending Hoare logic [39] with two operators, namely separation conjunction and separation implication; these operators are used to formulate assertions over disjoint parts of the heap. In [58], O'Hearn et al. define a logic for reasoning about programs that alter data structures; to this effect they define a low-level storage model based on a heap with associated access operations, along with axiomatizations for these operations. The resulting model supports local reasoning, whereby only those cells that a program accesses are referenced in specifications and proofs.
- *Reachability Predicates.* This approach defines and uses predicates that characterize reachable nodes in an arbitrary data structure [57]. Indexed predicate abstraction [43] and Boolean heaps [59] generalize the predicate abstraction domain so that it enables the inference of universally quantified invariants. In [36], Gulwani et al. show how to combine different abstract domains to obtain universally quantified domains that capture properties of linked lists. Craig interpolation has also been used to find universally quantified invariants for linked lists [47]. In [48], Mehta and Nipkow model heaps as mappings from addresses to values, and pointer structures are mapped to higher level data types for the verification of inductively defined data types like lists and trees. In [31], Filliatre and Marche introduce a method for proving that a program satisfies its specification and is free of null pointer referencing and out-of-bounds array access. Their approach is based on Burstall's model for structures extended to arrays and pointers. Similar tools have been developed for C-like languages, including Astrée [4], Caveat [66], and SDV [17], but they are bounded to specific provers. In [49, 50], Meyer presents a comprehensive theory for modeling pointer-rich object structures and proving their properties; the model proposed by Meyer comes in two versions, a coarse-grained version that supports the analysis of the overall properties of the object structures, and a fine-grained version, that analyzes object structures at the level of individual fields. Meyer's approach is represented in Eiffel syntax, and uses simple discrete mathematics.

Our interest in pointer semantics is much more recent than all these authors, and is driven by (and limited to) our interest in capturing conditions of abort avoidance as they pertain to illegal pointer references. Whereas we had thought initially that we could produce invariant relations that represent the scope equation of pointer references in loops for arbitrary data structures, we have subsequently resolved to generate invariant relations for well known data structures instead, for several reasons: First, generating invariant relations for the general case is very difficult; second, many authors whose work we have reviewed above appear to focus on well-known data structures rather than to arbitrary pointer-based structures; third, existing algorithms of shape analysis give us confidence that we can proceed by first analyzing the shape of our data, then deploying specialized invariant relations accoring to the shape that has been identified.

7.3. Assessment

Against the backdrop of all the approaches, methods and tools that we have surveyed above, we can characterize our work by the following premises:

- *The Study of Convergence as a Special Form of Functional Analysis.* Traditionally, the analysis of loop termination is studied separately from the analysis of its functional properties, with the latter relying on invariant assertions and the former relying on variant functions. By contrast, we use the same concept, namely invariant relations, to characterize the termination conditions and the functional properties of loops. From a conceptual viewpoint, we find it appealing to use the same approach/ means to analyze the function of the loop and the convergence condition of the loop, as the domain of a function is an integral part of the function, rather than an orthogonal attribute.

In practice, we find that when we select an invariant relation that is symmetric (in addition to being reflexive and transitive), we capture what we usually think of as functional properties (that arise in the study of partial correctness); on the other hand, when we select an invariant relation that is antisymmetric (in addition to being reflexive and transitive), we capture termination properties (that arise in the study of total correctness). The indiscriminate use of invariant relations subsumes traditional approaches seamlessly within the same framework.

- *We analyze termination and abort-freedom as instances of the same property, rather than two separate properties.* This is not merely a matter of semantics, but has concrete implications: Let R be an invariant relation that captures termination properties and Q be an invariant relation that captures abort-freedom properties. Treating these properties separately yields the following approximation of WL
$$R\overline{T} \cap Q\overline{T}$$
whereas treating them jointly yields the following approximation:
$$(R \cap Q)\overline{T}.$$
Because the relational product does not distribute over intersection, these expressions are not equivalent. In fact we have,
$$(R \cap Q)\overline{T} \subseteq R\overline{T} \cap Q\overline{T}.$$
Hence treating termination and abort-freedom jointly produces tighter approximations of WL than treating them separately. The example we show in section 1.2 illustrates this difference: whereas the termination condition deals exclusively in variable i and the abort-freedom condidtion deals exclusively in variable x, the condition that we generate with our approach captures the invariant relation that the loop maintains between i and x ($\{(s,s')|5i+2x=5i'+2x'\}$) and produces a more precise condition of convergence.
- *It is better to know something about every execution than to know everything about some executions.* Whereas other approaches approximate loops by setting an upper bound on their number of iterations, we approximate loops by means of invariant relations; invariant relations provide us with arbitrarily loose (or tight) approximations of (supersets of) the function of the loop. We argue that analyzing a limited number of executions may be insufficient and capturing all the functional detail of an execution may be unnecessary for a given verification goal. By contrast, invariant relations can be used be capture only enough information to meet our verification goals (sufficiency); and they approximate the function of the loop regardless of how many times it iterates (this is necessary if we want to draw conclusions that we can claim with certainty about all possible executions).
- *Linking the Convergence condition of a loop to the convergence condition of its loop body.* Taken together, theorems 2 and 3 (where we let D be BL) produce the following approximation of the convergence condition of a loop:
$$WL \subseteq \overline{(B'^{*}(B'^{+} \cap \overline{BL}))}\overline{T}.$$

In other words, they link the convergence condition of the loop (WL) to the convergence condition of its loop body (BL), for any relevant relation B'. This is interestring because it enables us to analyze nested loops by applying this result repeatedly, starting from the innermost loop and proceeding in a stepwise manner to compute the convergence condition of the outermost loop; this how we analyzed the nested loops of section 1.2.

In summary, we do not see our work as contributing a tool as much as it is contributing a basis for evolving a method that models termination and abort-freedom as complementary aspects of the same property, modeled in a seamless manner over a continuum.

Acknowledgements

The authors gratefully acknowledge the thoughtful, insightful and knowledgeable feedback they have received from the anonymous reviewers; their detailed feedback has helped the authors gain perspective and valuable new insights on their work. It has also helped them achieve vast improvements to the paper's presentation and content.

References

[1] Jose Bacelar Almeida, Maria Joao Frade, Jorge Sousa Pinto, and Simao Melo de Sousa. *Rigorous Software Development: An Introduction to Program Verification.* Springer Verlag, 2011.

[2] Corinne Ancourt, Fabien Coelho, and François Irigoin. A modular static analysis approach to affine loop invariants detection. *Electronic Notes on Theoretical Computer Science*, 267(1):3–16, 2010.

[3] Algirdas Avizienis, Jean Claude Laprie, Brian Randell, and Carl E Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.

[4] B. Blanchet, Patrick Cousot, Radhia Cousot, J. Feret, L. Mauborgne, A. Mine, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *Proceedings, PLDI 2003*, pages 196–207, San Diego, CA, USA, June 2003. ACM.

[5] R.S. Boyer and J.S. Moore. *A Computational Logic Handbook.* Academic Press inc., 1988.

[6] Aaron Bradley, Zohar Manna, and Henry Sipma. Linear ranking with reachability. In *Proceedings, Computer Aided Verification*, 2005.

[7] Aaron Bradley, Zohar Manna, and Henry Sipma. The polyranking principle. In *Proceedings, ICALP 2005*, 2005.

[8] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. Termination analysis of integer linear loops. In *CONCUR*, pages 488–502, 2005.

[9] Chris Brink, Wolfram Kahl, and Gunther Schmidt. *Relational Methods in Computer Science.* Advances in Computer Science. Springer Verlag, Berlin, Germany, 1997.

[10] David Buehler, Pascal Couq, Boris Yakobowski, Mathieu Lemerre, Andre Maroneze, Valentin Perrelle, and Virgile Prevosto. The eva plug-in: Sulfur 20171101. Technical report, CEA LIST, Software Reliability Laboratory, Saclay, F-91191, 2017.

[11] Jochen Burghardt and Hans Werner Pohl. An introduction to verifast for java. Technical report, Fraunhofer Institute, September 2014.

[12] Jacob Burnim, Nicholas Jalbert, Christos Stergiou, and Koushik Sen. Looper: Lightweight detection of infinite loops at runtime. In *ASE*, pages 161–169, 2009.

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

[14] Aziem Chawdhary, Byron Cook, Sumit Gulwani, Mooly Sagiv, and Hongseok Yang. Ranking abstractions. In *ESOP*, pages 148–162, 2008.

[15] Michael Colón and Henny Sipma. Practical methods for proving program termination. In *Proc. International Conference on Computer Aided Verification*, CAV '02, pages 442–454, London, UK, UK, 2002. Springer-Verlag.

[16] 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.

[17] Byron Cook. Static driver verifier. Technical report, Microsoft Inc., http://www.microsoft.com/whdc/devtools/, 2012.

[18] Byron Cook, Carsten Fuhs, Kaustubh Nimkar, and Peter O'Hearn. Disproving termination with overapproximation. In *Proceedings, FMCAD*, Lausanne, CH, October 2014.

[19] Byron Cook, Andreas Podelski, and Andrey 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.

[20] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Proving program termination. *Communications of the ACM*, 54(5), 2011.

[21] Byron Cook, Abigail See, and Florian Zuleger. Ramsey vs. lexicographic termination proving. In *Proceedings, TACAS 2013: 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 47–61. Springer Verlag, 2013.

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

[23] 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.

[24] Patrick Cousot. Abstract interpretation. Technical Report www.di.ens.fr/˜cousot/AI/, Ecole Normale Superieure, Paris, France, August 2008.

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

[26] Patrick Cousot, Radhia Cousot, Jérôme Feret, Antoine Miné, Laurent Mauborgne, David Monniaux, and Xavier Rival. Varieties of static analyzers: A comparison with astree. In *TASE*, pages 3–20, 2007.

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

[28] Vijay D'Silva and Caterina Urban. Complexity of bradley-manna-sipma lexicographic functions. In Daniel Kroening and Corina S Pasareanu, editors, *CAV 2015: Computer Aided Verification*, number 9206 in Lecture Notes in Computer Science, San Francisco, CA, USA, July, 18-24 2015.

[29] K. Durant, W. Visser, and C. Pasareanu. Investigating termination of affine loops with jpf. In *Java PathFinder Workshop*, Lawrence, KS, 2012.

[30] Stephan Falke, Deepak Kapur, and Carsten Sinz. Termination analysis of imperative programs using bitvector arithmetic. In *VSTTE*, pages 261–277, 2012.

[31] J. Filliatre and C. Marche. Multi prover verification of c programs. In *Procedings, ICFEM*, pages 15–29, 2004.

[32] Wided Ghardallou, Olfa Mraihi, Asma Louhichi, Lamia Labed Jilani, Khaled Bsaies, and Ali Mili. A versatile concept for the analysis of loops. *Journal of Logic and Algebraic Programming*, 81(5):606–622, May 2012.

[33] David Gries. *The Science of Programming.* Springer Verlag, 1981.

[34] David Gries and Fred B. Schneider. *A Logical Approach to Discrete Math.* Springer-Verlag, 1993.

[35] Bhargav S. Gulavani, Supratik Chakraborty, Ganesan Ramalingam, and Aditya V. Nori. Bottom up shape analysis using lisf. *ACM Transactions on Programming Languages and Systems*, 33(5), 2011.

[36] S. Gulwani, B. McCloskey, and A. Tiwari. Lifting abstract interpreters to quantified logic domains. In *35th ACM Symposium on Principles of Programming Languages*, pages 235–246. ACM, january 2008.

[37] Ashutosh Gupta, Thomas A. Henzinger, Rupak Majumdar, Andrey Rybalchenko, and Ru-Gang Xu. Proving non-termination. In *POPL*, pages 147–158, 2008.

[38] B. Hackett and A. Aiken. How is aliasing used in systems software. In *Proceedings, 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 69–80, 2006.

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

[40] Lamia Labed Jilani, Olfa Mraihi, Asma Louhichi, Wided Ghardallou, Khaled Bsaies, and Ali Mili. Invariant relations and invariant functions: An alternative to invariant assertions. *Journal of Symbolic Computation*, 48:1–36, May 2013.

[41] D. Kästner, S. Wilhelm, S. Nenova, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival. Astrée: Proving the absence of runtime errors. In *Embedded Real Time Software and Systems (ERTS^2 2010)*, pages 1–9, May 2010.

[42] 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.

[43] S.K. Lahiri and R.E. Bryant. Constructing quantified invariants via predicate abstraction. In *Proceedings, VMCAI*, pages 267–281, 2004.

[44] Wonchan Lee, Bow-Yaw Wang, and Kwangkeun Yi. Termination analysis with algorithmic learning. In *CAV*, pages 88–104, 2012.

[45] Glenn R. Luecke, James Coyle, Jim Hoekstra, Marina Kraeva, Olga Taborskaia, and Yanmei Wang. A survey of systems for detecting serial run-time errors. *Concurrency and Computation Practice and Experience*, 18:1885–1907, December 2006.

[46] Zohar Manna. *A Mathematical Theory of Computation.* McGraw-Hill, 1974.

[47] K. L. McMillan. Quantified invariant generation using an interpolating saturation prover. In *Proceedings, TACAS*, pages 413–427, 2008.

[48] F. Mehta and T. Nipkow. Proving pointer programs in higher order logic. *Inf. Comput.*, 199(1-2):200–277, 2005.

[49] Bertrand Meyer. Proving pointer program properties. part i: Context and overview. *Journal of Object Technology*, 2(2):87–108, 2003.

[50] Bertrand Meyer. Proving pointer program properties. part ii: The overall object structure. *Journal of Object Technology*, 2(3):77–100, 2003.

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

[52] Ali Mili, Shir Aharon, Chaitanya Nadkarni, Olfa Mraihi, Asma Louhichi, and Lamia Labed Jilani. Reflexive transitive invariant relations: A basis for computing loop functions. *Journal of Symbolic Computation*, 45:1114–1143, 2009.

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

[54] Harlan D. Mills, Victor R. Basili, John D. Gannon, and Dick R. Hamlet. *Structured Programming: A Mathematical Approach.* Allyn and Bacon, Boston, Ma, 1986.

[55] Olfa Mraihi, Asma Louhichi, Lamia Labed Jilani, Jules Desharnais, and Ali Mili. Invariant assertions, invariant relations, and invariant functions. *Science of Computer Programming*, 78(9):1212–1239, September 2013.

[56] S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufman, 1997.

[57] G. Nelson. Verifying reachability invariants of linked structures. In *Proceedings, POPL 1983: Principles of Programming Languages*, pages 38–47, 1983.

[58] P. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *Proceedings, CSL*, pages 1–19, 2001.

[59] A. Podelski and T. Wies. Boolean heaps. In *Procedings, SAS*, pages 267–282, 2005.

[60] Andreas Podelski and Andrey Rybalchenko. A complete method for the synthesis of linear ranking functions. In *VMCAI*, pages 239–251, 2004.

[61] Andreas Podelski and Andrey Rybalchenko. Transition invariants. In *LICS*, pages 32–41, 2004.

[62] Andreas Podelski and Andrey Rybalchenko. Transition invariants and transition predicate abstraction for program termination. In *TACAS*, pages 3–10, 2011.

[63] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings, LICS*, pages 55–74, 2002.

[64] S. Sagiv, T. W. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Logics and Systems*, 24(3):217–298, 2002.

[65] Fausto Spoto, Fred Mesnard, and Etienne Payet. A termination analyzerfor java bytecode based on path length. *ACM Transactions on Programming Languages and Systems*, 32(3), 2010.

[66] CAVEAT team. Caveat project. Technical report, Commissariat à l'Energie Atomique, http://www-drt.cea.fr/Pages/List/Ise/LSL/Caveat/, 2012.

[67] Ashish Tiwari. Termination of linear programs. In *CAV*, pages 70–82, 2004.

[68] Aliaksei Tsitovich, Natasha Sharygina, Christoph M. Wintersteiger, and Daniel Kroening. Loop summarization and termination analysis. In *Proc.International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 81–95, 2011.

[69] 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.

[70] Klaus Wissing. Static analysis of dunamic properties: Automatic program verification to prove the absence of dynamic runtime errors. In *Proceedings, GI Jahrestagung*, 2007.