Loop invariants: analysis, classification, and examples

Carlo A. Furia · Bertrand Meyer · Sergey Velder

Abstract

Software verification has emerged as a key concern for ensuring the continued progress of information technology. Full verification generally requires, as a crucial step, equipping each loop with a "loop invariant". Beyond their role in verification, loop invariants help program understanding by providing fundamental insights into the nature of algorithms. In practice, finding sound and useful invariants remains a challenge. Fortunately, many invariants seem intuitively to exhibit a common flavor. Understanding these fundamental invariant patterns could therefore provide help for understanding and verifying a large variety of programs.

We performed a systematic identification, validation, and classification of loop invariants over a range of fundamental algorithms from diverse areas of computer science. This article analyzes the patterns, as uncovered in this study, governing how invariants are derived from postconditions; it proposes a taxonomy of invariants according to these patterns, and presents its application to the algorithms reviewed. The discussion also shows the need for high-level specifications based on "domain theory". It describes how the invariants and the corresponding algorithms have been mechanically verified using an automated program prover; the proof source files are available. The contributions also include suggestions for invariant inference and for model-based specification.

Contents

1 Introduction: inductive invariants

The problem of guaranteeing program correctness remains one of the central challenges of software engineering, of considerable importance to the information technology industry and to society at large, which increasingly depends, for almost all of its processes, on correctly functioning programs. As defined by Tony Hoare [32], the "Grand Challenge of Program Verification" mobilizes many researchers and practitioners using a variety of techniques.

Some of these techniques, such as model checking [10] and abstract interpretation [13], are directed at finding specific errors, such as the possible violation of a safety property. An advantage of these techniques is that they work on programs as they are, without imposing a significant extra annotation effort on programmers. For full functional correctness—the task of proving that a program satisfies a complete specification—the approach of choice remains, for imperative programs, the Floyd-Hoare-Dijkstra style of axiomatic semantics. In this approach, programs must be equipped with annotations in the form of *assertions*. Every loop, in particular, must have a *loop invariant*.

Finding suitable loop invariants is a crucial and delicate step to verification. Although some programmers may see invariant elicitation as a chore needed only for formal verification, the concept is in fact widely useful, including for informal development: the invariant gives fundamental information about a loop, showing what it is trying to achieve and how it achieves it, to the point that (in some people's view at least) it is impossible to understand a loop without knowing its invariant.

To explore and illustrate this view, we have investigated a body of representative loop algorithms in several areas of computer science, to identify the corresponding invariants, and found that they follow a set of standard patterns. We set out to uncover, catalog, classify, and verify these patterns, and report our findings in the present article.

Finding an invariant for a loop is traditionally the responsibility of a human: either the person performing the verification, or the programmer writing the loop in the first place (a better solution, when applicable, is the *constructive* approach to programming advocated by Dijkstra and others [17, 26, 46]). More recently, techniques have been developed for *inferring* invariants automatically, or semi-automatically with some human help (we review them in Section 5). We hope that the results reported here will be useful in both cases: for humans, to help obtain the loop invariants of new or existing programs, a task that many programmers still find challenging; and for invariant inference tools.

For all algorithms presented in the paper[1], we wrote fully annotated implementations and processed the result with the Boogie program verifier [43], providing proofs of correctness. The Boogie implementations are available at:[2]

[1] With the exception of those in Sections 4.5 and 4.7, whose presentation is at a higher level of abstraction, so that a complete formalization would have required complex axiomatization of geometric and numerical properties beyond the focus of this paper.

[2] In the repository, the branch `inv_survey` contains only the algorithms described in the paper; see `http://goo.gl/DsdrV` for instruction on how to access it.

`http://bitbucket.org/sechairethz/verified/`

This verification result reinforces the confidence in the correctness of the algorithms presented in the paper and their practical applicability.

The rest of this introductory section recalls the basic properties of invariants. Section 2 introduces a style of expressing invariants based on "domain theory", which can often be useful for clarity and expressiveness. Section 3 presents two independent classifications of loop invariant clauses, according to their role and syntactic similarity with respect to the postcondition. Section 4 presents 21 algorithms from various domains; for each algorithm, it presents an implementation in pseudo-code annotated with complete specification and loop invariants. Section 5 discusses some related techniques to infer invariants or other specification elements automatically. Section 6 draws lessons from the verification effort. Section 7 concludes.

1.1 Loop invariants basics

The loop invariants of the axiomatic approach go back to Floyd [20] and Hoare [30] (see Hatcliff et al. [28] for a survey of notations for and variants of the fundamental idea). For this approach and for the present article, a loop invariant is not just a quantity that remains unchanged throughout executions of the loop body (a notion that has also been studied in the literature), but more specifically an "inductive invariant", of which the precise definition appears next. Program verification also uses other kinds of invariant, notably class invariants [31, 47], which the present discussion surveys only briefly in Section 1.4.

The notion of loop invariant is easy to express in the following loop syntax taken from Eiffel:

```
1   from
2       Init
3   invariant
4       Inv
5   until
6       Exit
7   variant
8       Var
9   loop
10      Body
11  end
```

(the **variant** clause helps establish termination as discussed later). *Init* and *Body* are each a compound (a list of instructions to be executed in sequence); either or both can be empty, although *Body* normally will not. *Exit* and *Inv* (the inductive invariant) are both Boolean expressions, that is to say, predicates on the program state. The semantics of the loop is:

1. Execute *Init*.
2. Then, if *Exit* has value **True**, do nothing; if it has value **False**, execute *Body*, and repeat step 2.

Another way of stating this informal specification is that the execution of the loop body consists of the execution of *Init* followed by zero or more executions of *Body*, stopping as soon as *Exit* becomes **True**.

There are many variations of the loop construct in imperative programming languages: "while" forms which use a continuation condition rather than the inverse exit condition; "do-until" forms that always execute the loop body at least once, testing for the condition at the end rather than on entry; "for" or "do" forms ("**across**" in Eiffel) which iterate over an integer interval or a data structure. They can all be derived in a straightforward way from the above basic form, on which we will rely throughout this article.

The invariant *Inv* plays no direct role in the informal semantics, but serves to reason about the loop and its correctness. *Inv* is a correct invariant for the loop if it satisfies the following conditions:

1. Every execution of *Init*, started in the state preceding the loop execution, will yield a state in which *Inv* holds.

2. Every execution of *Body*, started in any state in which *Inv* holds and *Exit* does not hold, will yield a state in which *Inv* holds again.

If these properties hold, then any terminating execution of the loop will yield a state in which both *Inv* and *Exit* hold. This result is a consequence of the loop semantics, which defines the loop execution as the execution of *Init* followed by zero or more executions of *Body*, each performed in a state where *Exit* does not hold. If *Init* ensures satisfaction of the invariant, and any one execution of *Body* preserves it (it is enough to obtain this property for executions started in a state not satisfying *Exit*), then *Init* followed by *any* number of executions of *Body* will.

Formally, the following classic inference rule [31, 47] uses the invariant to express the correctness requirement on any loop:

$$\frac{\{P\}\ \mathit{Init}\ \{\mathit{Inv}\}, \quad \{\mathit{Inv} \wedge\ \neg\ \mathit{Exit}\}\ \mathit{Body}\ \{\mathit{Inv}\}}{\{P\}\ \textbf{from}\ \mathit{Init}\ \textbf{until}\ \mathit{Exit}\ \textbf{loop}\ \mathit{Body}\ \textbf{end}\ \{\mathit{Inv} \wedge \mathit{Exit}\}}.$$

This is a partial correctness rule, useful only for loops that terminate. Proofs of termination are in general handled separately through the introduction of a loop variant: a value from a well-founded set, usually taken to be the set of natural numbers, which decreases upon each iteration (again, it is enough to show that it does so for initial states not satisfying *Exit*). Since in a well-founded set all decreasing sequences are finite, the existence of a variant expression implies termination. The rest of this discussion concentrates on the invariants; it only considers terminating algorithms, of course, and includes the corresponding **variant** clauses, but does not explain why the corresponding expression are indeed loop variants (non-negative and decreasing). Invariants, however, also feature in termination proofs, where they ensure that the variant ranges over a well-founded set (or, equivalently, the values it takes are bounded from below).

If a loop is equipped with an invariant, proving its partial correctness means establishing the two hypotheses in the above rules:

- $\{P\}$ *Init* $\{Inv\}$, stating that the initialization ensures the invariant, is called the *initiation* property.
- $\{Inv \wedge \neg Exit\}$ *Body* $\{Inv\}$, stating that the body preserves the invariant, is called the *consecution* (or *inductiveness*) property.

1.2 A constructive view

We may look at the notion of loop invariant from the constructive perspective of a programmer directing his or her program to reach a state satisfying a certain desired property, the postcondition. In this view, program construction is a form of problem-solving, and the various control structures are problem-solving techniques [17, 46, 26, 48]; a loop solves a problem through successive approximation.

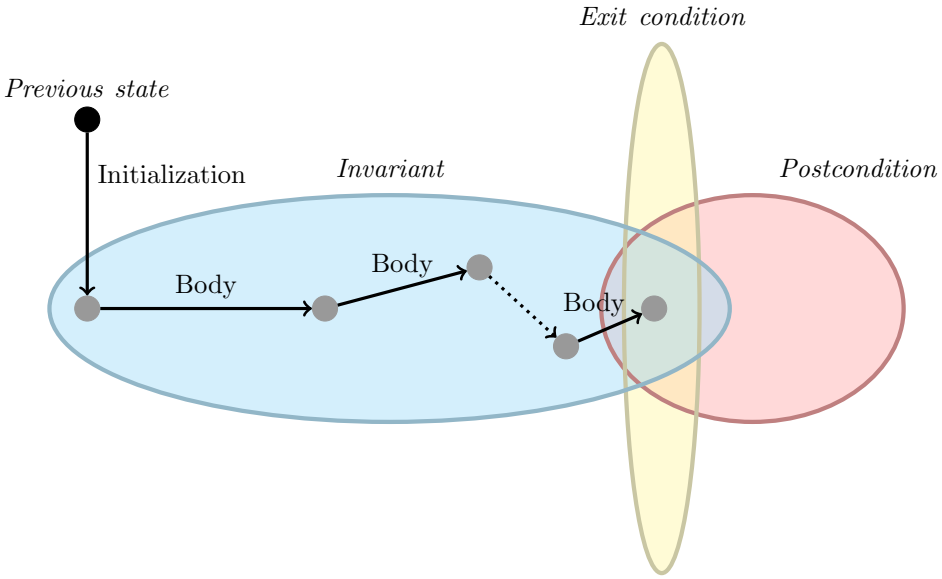


Figure 1: The loop as a computation by approximation.

The idea of this solution, illustrated by Figure 1, is the following:

- Generalize the postcondition (the characterization of possible solutions) into a broader condition: the invariant.
- As a result, the postcondition can be defined as the combination ("and" in logic, intersection in the figure) of the invariant and another condition: the exit condition.
- Find a way to reach the invariant from the previous state of the computation: the initialization.

- Find a way, given a state that satisfies the invariant, to get to another state, still satisfying the invariant but closer, in some appropriate sense, to the exit condition: the body.

For the solution to reach its goal after a finite number of steps we need a notion of discrete "distance" to the exit condition. This is the loop variant.

The importance of the above presentation of the loop process is that it highlights the nature of the invariant: it is a generalized form of the desired postcondition, which in a special case (represented by the exit condition) will give us that postcondition. This view of the invariant, as a particular way of generalizing the desired goal of the loop computation, explains why the loop invariant is such an important property of loops; one can argue that understanding a loop means understanding its invariant (in spite of the obvious observation that many programmers write loops without ever formally learning the notion of invariant, although we may claim that if they understand what they are doing they are relying on some intuitive understanding of the invariant anyway, like Molière's Mr. Jourdain speaking in prose without knowing it).

The key results of this article can be described as generalization strategies to obtain invariants from postconditions.

1.3 A basic example

To illustrate the above ideas, the 2300-year-old example of Euclid's algorithm, while very simple, is still a model of elegance. The postcondition of the algorithm is

$$\textbf{Result} = \gcd(a, b),$$

where the positive integers a and b are the input and gcd is the mathematical Greatest Common Divisor function. The generalization is to replace this condition by

$$\textbf{Result} = x \quad \wedge \quad \gcd(\textbf{Result}, x) = \gcd(a, b) \tag{1}$$

with a new variable x, taking advantage of the mathematical property that, for every x,

$$\gcd(x, x) = x. \tag{2}$$

The second conjunct, a generalization of the postcondition, will serve as the invariant; the first conjunct will serve as the exit condition. To obtain the loop body we take advantage of another mathematical property: for every $x > y$,

$$\gcd(x, y) = \gcd(x - y, y), \tag{3}$$

yielding the well-known algorithm in Figure 2. (As with any assertion, writing clauses successively in the invariant is equivalent to a logical conjunction.) This form of Euclid's algorithm uses subtraction; another form, given in Section 4.2.2, uses integer division.

We may use this example to illustrate some of the orthogonal categories in the classification developed in the rest of this article:

```
1  from
2    Result := a ; x := b
3  invariant
4    Result >0
5    x >0
6    gcd (Result, x) = gcd (a, b)
7  until
8    Result = x
9  loop
10   if Result >x then
11     Result := Result − x
12   else      −− Here x is strictly greater than Result
13     x := x − Result
14   end
15 variant
16   max (Result, x)
17 end
```

Figure 2: Greatest common divisor with substraction.

- The last clause of the invariant is an *essential invariant*, representing a weakening of the postcondition. The first two clauses are a *bounding invariant*, indicating that the state remains within certain general boundaries, and ensuring that the "essential" part is defined.
- The essential invariant is a *conservation invariant*, indicating that a certain quantity remains equal to its original value.
- The strategy that leads to this conservation invariant is *uncoupling*, which replaces a property of one variable (**Result**), used in the postcondition, by a property of two variables (**Result** and x), used in the invariant.

The proof of correctness follows directly from the mathematical property stated: (2) establishes initiation, and (3) establishes consecution.

Section 4.2.2 shows how the same technique is applicable backward, to guess likely loop invariants given an algorithm annotated with pre- and postcondition: mutating the latter yields a suitable loop invariant.

1.4 Other kinds of invariant

Loop invariants are the focus of this article, but before we return to them it is useful to list some other kinds of invariant encountered in software. (Yet other invariants, which lie even further beyond the scope of this discussion, play fundamental roles in fields such as physics; consider for example the invariance of the speed of light under a Lorentz transformation, and of time under a Galilean transformation.)

In object-oriented programming, a class invariant (also directly supported by the Eiffel notation [18]) expresses a property of a class that:

- Every instance of the class possesses immediately after creation, and
- Every exported feature (operation) of the class preserves,

with the consequence that whenever such an object is accessible to the rest of the software it satisfies the invariant, since the life of an object consists of creation followed by any number of "qualified" calls $x.f$ to exported features f by clients of the class. The two properties listed are strikingly similar to initiation and consecution for loop invariants, and the connection appears clearly if we model the life of an object as a loop:

```
1 from
2   create x.make −− Written in some languages as x := new C()
3 invariant
4   CI          −− The class invariant
5 until
6   "x is no longer needed"
7 loop
8   x. some_feature_of_the_class
9 end
```

Also useful are Lamport-style invariants [41] used to reason about concurrent programs, which obviate the need for the "ghost variables" of the Owicki-Gries method [50]). Like other invariants, a Lamport invariant is a predicate on the program state; the difference is that the definition of the states involves not only the values of the program's variables but also the current point of the execution of the program ("Program Counter" or PC) and, in the case of a concurrent program, the collection of the PCs of all its concurrent processes. An example of application is the answer to the following problem posed by Lamport [42].

> Consider N processes numbered from 0 through $N-1$ in which each process i executes
>
> $$\begin{aligned} \ell_0^i: &\quad x[i] := 1 \\ \ell_1^i: &\quad y[i] := x[(i-1) \bmod N] \\ \ell_2^i: & \end{aligned}$$
>
> and stops, where each $x[i]$ initially equals 0. (The reads and writes of each $x[i]$ are assumed to be atomic.) [...] The algorithm [...] maintains an inductive invariant. Do you know what that invariant is?

If we associate a proposition $@(m, i)$ for $m = 1, 2, 3$ that holds precisely when the execution of process i reaches location ℓ_m^i, an invariant for the algorithm

can be expressed as:

$$@(2, i) \Longrightarrow \left(\begin{array}{c} @(0, (i-1) \bmod N) \wedge y[i] = 0 \\ \vee \\ @(1, (i-1) \bmod N) \wedge y[i] = 1 \\ \vee \\ @(2, (i-1) \bmod N) \wedge y[i] = 1 \end{array} \right).$$

Yet another kind of invariant occurs in the study of dynamical systems, where an invariant is a region $I \subseteq \mathbb{S}$ of the state space $\mathbb{S}$ such that any trajectory starting in I or entering it stays in I indefinitely in the future:

$$\forall x \in I, \forall t \in \mathbb{T} : \Phi(t, x) \in I,$$

where $\mathbb{T}$ is the time domain and $\Phi : \mathbb{T} \times \mathbb{S} \to \mathbb{S}$ is the evolution function. The connection between dynamical system invariants and loop invariants is clear in the constructive view (Section 1.2), and can be formally derived by modeling programs as dynamical systems or using some other operational formalism [22]. The differential invariants introduced in the study of hybrid systems [54] are also variations of the invariants defined by dynamical systems.

2 Expressing invariants: domain theory

To discuss and compare invariants we need to settle on the expressiveness of the underlying invariant language: what do we accept as a loop invariant?

The question involves general assertions, not just invariants; more generally, we must make sure that any convention for invariants is compatible with the general scheme used for pre/post specification, since an invariant is a mutation (possibly a weakening) of the postcondition.

The mathematical answer to the basic question is simple: an assertion other than a routine postcondition, in particular a loop invariant, is a *predicate* on the program state. For example, the assertion $x > 0$, where x is a program variable, is the predicate that holds of all computation states in which the value of that variable is positive. (Another view would consider it as the *subset* of the state space containing all states that satisfy the condition; the two views are equivalent since the predicate is the characteristic function of the subset, and the subset is the inverse domain of "true" for the predicate.)

A routine postcondition is usually a predicate on *two* states, since the specification of a routine generally relates new values to original ones. For example, an increment routine yields a state in which the counter's value is one more on exit than on entry. The **old** notation, available for postconditions in Eiffel and other programming languages supporting contracts, reflects this need; for example, a postcondition clause could read *counter* = **old** *counter* + 1. Other notations, notably the Z specification language [60], have a notation for "new" rather than "old", as in *counter*' = *counter* + 1 where the primed variable denotes the new value. Although invariants are directly related to postconditions,

we will be able in this discussion to avoid such notations and treat invariants as one-state functions. (Technically, this is always possible by recording the entry value as part of the state.)

Programming languages offer a mechanism directly representing predicates on states: Boolean expressions. This construct can therefore be used—as in the $x > 0$ example—to represent assertions; this is what assertion-aware programming languages typically do, often extending it with special notations such as **old** and support for quantifiers.

This basic language decision leaves open the question of the *level of expressiveness* of assertions. There are two possibilities:

- Allow assertions, in particular postconditions and loop invariants, to use *functions* and *predicates* defined using some appropriate mechanism (often, the programming language's function declaration construct) to express high-level properties based on a domain theory covering specifics of the application area. We call this approach *domain theory*.[3]
- Disallow this possibility, requiring assertions always to be expressed in terms of the constructs of the assertion language, without functions. We call this approach *atomic assertions*.

The example of Euclid's algorithm above, simple as it is, was already an example of the domain-theory-based approach because of its use of a function gcd in the invariant clause

$$\gcd(\mathbf{Result}, x) = \gcd(a, b) \tag{4}$$

corresponding to a weakening of the routine postcondition

$$\mathbf{Result} = \gcd(a, b).$$

It is possible to do without such a function by going back to the basic definition of the greatest common denominator. In such an atomic-assertion style, the postcondition would read

$\mathbf{Result} > 0$	(Alternatively, $\mathbf{Result} \geq 1$)
$a \setminus\setminus \mathbf{Result} = 0$	(**Result** divides a)
$b \setminus\setminus \mathbf{Result} = 0$	(**Result** divides b)
$\forall i \in \mathbb{N}$: $(a \setminus\setminus i = 0) \wedge (b \setminus\setminus i = 0)$ **implies** $i \leq \mathbf{Result}$	(**Result** is the greatest of all the numbers that satisfy the preceding properties).

Expressing the invariant in the same style requires several more lines since the definition of the greatest common divisor must be expanded for both sides of (4).

Even for such a simple example, the limitations of the atomic-assertion style are clear: because it requires going back to basic logical constructs every time, it does not scale.

[3] No relation with the study of partially ordered sets, also called domain theory [1].

Another example where we can contrast the two styles is any program that computes the maximum of an array. In the atomic-assertion style, the postcondition will be written

$\forall k \in \mathbb{Z}$: $a.lower \leq k \leq a.upper$ **implies** $a[k] \leq$ **Result** (Every element between bounds has a value smaller than **Result**)

$\exists k \in \mathbb{Z}$: $a.lower \leq k \leq a.upper \wedge a[k] =$ **Result** (Some element between bounds has the value **Result**).

This property is the definition of the maximum and hence needs to be written somewhere. If we define a function "max" to capture this definition, the specification becomes simply

$$\textbf{Result} = \max(a).$$

The difference between the two styles becomes critical when we come to the invariant of programs computing an array's maximum. Two different algorithms appear in Section 4.1. The first (Section 4.1.1) is the most straightforward; it moves an index i from $a.lower + 1$ to $a.upper$, updating **Result** if the current value is higher than the current result (initialized to the first element $a\ [a.lower]$). With a domain theory on arrays, the function max will be available as well as a notion of slice, where the slice $a\ [i\,..\,j]$ for integers i and j is the array consisting of elements of a in the range $[i, j]$. Then the invariant is simply

$$\textbf{Result} = \max(a\ [a.lower..\ i]),$$

which is ensured by initialization and, on exit when $i = a.upper$, yields the postcondition **Result** $= \max(a)$ (based on the domain-theory property that $a\ [a.lower..\ a.upper] = a$). The atomic-assertion invariant would be a variation of the expanded postcondition:

$$\begin{array}{l} \forall k \in \mathbb{Z}\text{: } a.lower \leq k \leq i \textbf{ implies } a[k] \leq \textbf{Result} \\ \exists k \in \mathbb{Z}\text{: } a.lower \leq k \leq i \wedge a[k] = \textbf{Result}. \end{array}$$

Consider now a different algorithm for the same problem (Section 4.1.2), which works by exploring the array from both ends, moving the left cursor i up if the element at i is less than the element at j and otherwise moving the right cursor j down. The atomic-assertion invariant can be written with an additional level of quantification:

$$\exists m : \left(\begin{array}{lcl} \forall k \in \mathbb{Z} : & a.lower \leq k \leq a.upper & \textbf{implies} \quad a[k] \leq m \\ \exists k \in \mathbb{Z} : & i \leq k \leq j & \textbf{and} \quad a[k] = m \end{array} \right). \tag{5}$$

Alternatively, we can avoid quantifier alternation using the characterization based on the complement property that the maximal element is *not* outside the slice $a[\,i\,..\,j\,]$:

$$\forall k \in \mathbb{Z} : \quad a.lower \leq k < i \vee j < k \leq a.upper \implies a[k] \leq a[\,i\,] \vee a[k] \leq a[j]. \tag{6}$$

The form without quantifier alternation is more amenable to automated reasoning, but it has the disadvantage that it requires additional ingenuity and is not a straightforward modification of the invariant for the one-way version of the algorithm. More significantly for this paper's point of view, both formulations (5)–(6) give an appearance of complexity even though the invariant is conceptually very simple, capturing in a nutshell the essence of the algorithm (as noted earlier, one of the applications of a good invariant is that it enables us to understand the core idea behind a loop):

$$\max(a) = \max(a[i..j]). \tag{7}$$

In words: the maximum of the entire array is to be found in the slice that has not been explored yet. On exit, where $i = j$, we are left with a one-element slice, whose value (this is a small theorem of the corresponding domain theory) is its maximum and hence the maximum of the whole array. The domain-theory invariant makes the algorithm and its correctness immediately clear.

The domain-theory approach means that, before any attempt to reason about an algorithm, we should develop an appropriate model of the underlying domain, by defining appropriate concepts such as greatest common divisor for algorithms on integers and slices and maximum for algorithms on arrays, establishing the relevant theorems (for example that $x > y \Longrightarrow \gcd(x, y) = \gcd(x - y, y)$ and that $\max(a[i..i]) = a[i]$). These concepts and theorems need only be developed once for every application domain of interest, not anew for every program over that domain. The programs can then use the corresponding functions in their assertions, in particular in the loop invariants.

The domain-theory approach takes advantage of standard abstraction mechanism of mathematics. Its only practical disadvantage, for assertions embedded in a programming language, is that the functions over a domain (such as gcd) must come from some library and, if themselves written in the programming language, must satisfy strict limitations; in particular they must be "pure" functions defined without any reference to imperative constructs. This issue only matters, however, for the practical embedding of invariants in programs; it is not relevant to the conceptual discussion of invariants, independent of any implementation concerns, which is the focus of this paper.

For the same reason, this paper does not explore—except for Section 6—the often delicate trade-off between succinctness of expression and amenability to automated reasoning. For example, the invariant (5) is concisely captured as (7) in domain-theory form even if it uses quantifier alternation; the different formulation (6) is not readily expressible in terms of slice and maximum functions, but it may be easier to handle by automatic theorem provers since complexity grows with quantifier alternation [51]. This paper's focus is on developing and understanding the essence of algorithms through loop invariants presented at the right level of abstraction, largely independent of the requirements posed by automated reasoning. Section 6, however, demonstrates that the domain-theory approach is still practically applicable.

The remainder of this article relies, for each class of algorithms, on the appropriate domain theory, whose components (functions and theorems) are

summarized at the beginning of the corresponding section. We will make no further attempt at going back to the atomic-assertion style; the examples above should suffice to show how much simplicity is gained through this policy.

3 Classifying invariants

Loop invariants and their constituent clauses can be classified along two dimensions:

- By their role with respect to the postcondition (Section 3.1), leading us to distinguish between "essential" and "bounding" invariant properties.
- By the transformation technique that yields the invariant from the postcondition (Section 3.2). Here we have techniques such as uncoupling and constant relaxation.

3.1 Classification by role

In the typical loop strategy described in Section 1.2, it is essential that successive iterations of the loop body remain in the convergence regions where the generalized form of the postcondition is defined. The corresponding conditions make up the *bounding invariant*; the clauses describing the generalized postcondition is the *essential invariant*. The bounding invariant for the greatest common divisor algorithm consists of the clauses

$$\begin{aligned} &\mathbf{Result} > 0 \\ &x > 0. \end{aligned}$$

The essential clause is

$$\gcd(\mathbf{Result}, x) = \gcd(a, b),$$

yielding the postcondition when $\mathbf{Result} = x$.

For the one-way maximum program, the bounding invariant is

$$a.lower \leq i \leq a.upper$$

and the essential invariant is

$$\mathbf{Result} = \max\,(a\ [a.lower .. i]),$$

yielding the postcondition when $i = a.upper$. Note that the essential invariant would not be defined without the bounding invariant, since the slice $a\,[1..\,i]$ would be undefined (if $i > a.upper$) or would be empty and have no maximum (if $i < a.lower$).

For the two-way maximum program, the bounding invariant is

$$a.lower \leq i \leq j \leq a.upper$$

and the essential invariant is

$$\max(a) = \max(a[i..j]),$$

yielding the postcondition when $i = j$. Again, the essential invariant would not be always defined without the bounding invariant.

The separation between bounding and essential invariants is often straightforward as in these examples. In case of doubt, the following observation will help distinguish. The functions involved in the invariant (and often, those of the postcondition) are often partial; for example:

- $\gcd(u, v)$ is only defined if u and v are both non-zero (and, since we consider natural integers only in the example, positive).
- For an array a and an integer i, $a[\,i\,]$ is only defined if $i \in [a.lower\,..\,a.upper]$, and the slice $a[\,i\,..\,j\,]$ is non-empty only if $[\,i\,..\,j\,] \subseteq [a.lower\,..\,a.upper]$.
- $\max(a)$ is only defined if the array a is not empty.

Since the essential clauses, obtained by postcondition generalization, use $\gcd(\textbf{Result}, x)$ and (in the array algorithms) array elements and maxima, the invariants must include the bounding clauses as well to ensure that these essential clauses are meaningful. A similar pattern applies to most of the invariants studied later.

3.2 Classification by generalization technique

The essential invariant is a mutation (often, a weakening) of the loop's postcondition. The following mutation techniques are particularly common:

Constant relaxation: replace a constant n (more generally, an expression which does not change during the execution of the algorithm) by a variable i, and use $i = n$ as part or all of the exit condition.

Constant relaxation is the technique used in the one-way array maximum computation, where the constant is the upper bound of the array. The invariant generalizes the postcondition "**Result** is the maximum of the array up to $a.lower$", where $a.lower$ is a constant, with "**Result** is the maximum up to i". This condition is trivial to establish initially for a non-empty array (take i to be $a.lower$), easy to extend to an incremented i (take **Result** to be the greater of its previous value and $a[\,i\,]$), and yields the postcondition when i reaches $a.upper$. As we will see in Section 4.1.4, binary search differs from sequential search by applying double constant relaxation, to both the lower and upper bounds of the array.

Uncoupling: replace a variable v (often **Result**) by two (for example **Result** and x), using their equality as part or all of the exit condition.

Uncoupling is used in the greatest common divisor algorithm.

Term dropping: remove a subformula (typically a conjunct), which gives a straightforward weakening.

Term dropping is used in the partitioning algorithm (Section 4.3.1).

Aging: replace a variable (more generally, an expression) by an expression that represents the value the variable had at previous iterations of the loop.

Aging typically accommodates "off-by-one" discrepancies between when a variable is evaluated in the invariant and when it is updated in the loop body.

Backward reasoning: compute the loop's postcondition from another assertion by backward substitution.

Backward reasoning can be useful for nested loops, where the inner loop's postcondition can be derived from the outer loop's invariant.

4 The invariants of important algorithms

The following subsections include a presentation of several algorithms, their loop invariants, and their connection with each algorithm's postcondition. Table 1 lists the algorithms and their category. For more details about variants of the algorithms and their implementation, we refer to standard textbooks on algorithms [45, 12, 37].

Table 1: The algorithms presented in Section 4.

Algorithm	Type	Section
Maximum search (one variable)	searching	§ 4.1.1
Maximum search (two variable)	searching	§ 4.1.2
Sequential search in unsorted array	searching	§ 4.1.3
Binary search	searching	§ 4.1.4
Integer division	arithmetic	§ 4.2.1
Greatest common divisor (with division)	arithmetic	§ 4.2.2
Exponentiation (by squaring)	arithmetic	§ 4.2.3
Long integer addition	arithmetic	§ 4.2.4
Quick sort's partitioning	sorting	§ 4.3.1
Selection sort	sorting	§ 4.3.2
Insertion sort	sorting	§ 4.3.3
Bubble sort (basic)	sorting	§ 4.3.4
Bubble sort (improved)	sorting	§ 4.3.5
Comb sort	sorting	§ 4.3.6
Knapsack with integer weights	dynamic programming	§ 4.4.1
Levenstein distance	dynamic programming	§ 4.4.2
Rotating calipers algorithm	computational geometry	§ 4.5
List reversal	data structures	§ 4.6.1
Binary search trees	data structures	§ 4.6.2
PageRank algorithm	fixpoint	§ 4.7

4.1 Array searching

Many algorithmic problems can be phrased as *search* over data structures—from the simple arrays up to graphs and other sophisticated representations. This section illustrates some of the basic algorithms operating on arrays.

4.1.1 Maximum: one-variable loop

The following routine *max_one_way* returns the maximum element of an unsorted array a of bounds $a.lower$ and $a.upper$. The maximum is only defined for a non-empty array, thus the precondition $a.count \geq 1$. The postcondition can be written

$$\textbf{Result} = \max(a).$$

Writing it in slice form, as **Result** $=\max(a\ [a.lower..a.upper])$ yields the invariant by constant relaxation of either of the bounds. We choose the second one, $a.upper$, yielding the essential invariant clause

$$\textbf{Result} = \max(a\ [a.lower..\ i]).$$

Figure 3 shows the resulting implementation of the algorithm.

```
1  max_one_way (a: ARRAY [T]): T
2    require
3      a.count ≥ 1   -- a.count is the number of elements of the array
4    local
5      i: INTEGER
6    do
7      from
8        i := a.lower ; Result := a [a.lower]
9      invariant
10       a.lower ≤ i ≤ a.upper
11       Result = max (a [a.lower, i])
12     until
13       i = a.upper
14     loop
15       i := i + 1
16       if Result < a [i] then Result := a [i] end
17     variant
18       a.upper − i + 1
19     end
20   ensure
21     Result = max (a)
22   end
```

Figure 3: Maximum: one-variable loop.

Proving initiation is trivial. Consecution relies on the domain-theory property that

$$\max(a\ [1..\ i{+}1]) = \max(\max(a\ [1..\ i]), a\ [i\ +\ 1]).$$

4.1.2 Maximum: two-variable loop

The one-way maximum algorithm results from arbitrarily choosing to apply constant relaxation to either $a.lower$ or (as in the above version) $a.upper$. Guided by a symmetry concern, we may choose double constant relaxation, yielding another maximum algorithm *max_two_way* which traverses the array from both ends. If i and j are the two relaxing variables, the loop body either increases i or decreases j. When $i = j$, the loop has processed all of a, and hence i and j indicate the maximum element.

The specification (precondition and postcondition) is the same as for the previous algorithm. Figure 4 shows an implementation.

```
1  max_two_way (a: ARRAY [T]): T
2    require
3      a.count ≥ 1
4    local
5      i, j: INTEGER
6    do
7      from
8        i := a.lower ;  j := a.upper
9      invariant
10       a.lower ≤ i ≤ j ≤ a.upper
11       max (a [i..j]) = max (a)
12     until
13       i = j
14     loop
15       if a [i] > a [j] then j := j − 1 else i := i + 1 end
16     variant
17       j − i
18     end
19     Result := a [i]
20   ensure
21     Result = max (a)
22   end
```

Figure 4: Maximum: two-variable loop.

It is again trivial to prove initiation. Consecution relies on the following two domain-theory properties:

$$j > i \ \wedge \ a\,[i] \geq a\,[j] \quad \Longrightarrow \quad \max(a\,[i\,..\,j]) = \max(a\,[i\,..\,j-1]) \qquad (8)$$

$$i < j \ \wedge \ a\,[j] \geq a\,[i] \quad \Longrightarrow \quad \max(a\,[i\,..\,j]) = \max(a\,[i+1..j]). \qquad (9)$$

4.1.3 Search in an unsorted array

The following routine *has_sequential* returns the position of an occurrence of an element *key* in an array a or, if *key* does not appear, a special value. The algorithm applies to any sequential structure but is shown here for arrays. For

simplicity, we assume that the lower bound $a.lower$ of the array is 1, so that we can choose 0 as the special value. Obviously this assumption is easy to remove for generality: just replace 0, as a possible value for **Result**, by $a.lower - 1$.

The specification may use the domain-theory notation *elements* (a) to express the set of elements of an array a. A simple form of the postcondition is

$$\textbf{Result} \neq 0 \iff key \in elements(a), \tag{10}$$

which just records whether the *key* has been found. We will instead use a form that also records where the element appears if present:

$$\textbf{Result} \neq 0 \implies key = a\ [\textbf{Result}] \tag{11}$$
$$\textbf{Result} = 0 \implies key \notin elements\ (a), \tag{12}$$

to which we can for clarity prepend the bounding clause

$$\textbf{Result} \in [0..\ a.upper]$$

to make it explicit that the array access in (11) is defined when needed.

If in (12) we replace a by $a\ [1..\ a.upper]$, we obtain the loop invariant of sequential search by constant relaxation: introducing a variable i to replace either of the bounds 1 and $a.upper$. Choosing the latter yields the following essential invariant:

$$\begin{array}{lcl} \textbf{Result} \in [0, i] & & \\ \textbf{Result} \neq 0 & \implies & key = a\ [\textbf{Result}] \\ \textbf{Result} = 0 & \implies & key \notin elements\ (a\ \ [1..\ i]), \end{array}$$

leading to an algorithm that works on slices $[1..\ i]$ for increasing i, starting at 0 and with bounding invariant $0 \leq i \leq a.count$, as shown in Figure 5.[4]

To avoid useless iterations the exit condition may be replaced by $i = a.upper \vee \textbf{Result} > 0$.

To prove initiation, we note that initially **Result** is 0 and the slice $a\ \ [1..\ i]$ is empty. Consecution follows from the domain-theory property that, for all $1 \leq i < a.upper$:

$$key \in elements(a\ \ [1..\ i{+}1]) \iff key \in elements(a\ \ [1..\ i\,]) \vee key = a\ [\,i{+}1].$$

4.1.4 Binary search

Binary search works on sorted arrays by iteratively halving a segment of the array where the searched element may occur. The search terminates either when the element is found or when the segment becomes empty, implying that the element appears nowhere in the array.

As already remarked by Knuth many years ago [37, Vol. 3, Sec. 6.2.1]:

[4] Note that in this example it is OK for the array to be empty, so there is no precondition on $a.upper$, although general properties of arrays imply that $a.upper \geq 0$; the value 0 corresponds to an empty array.

```
1  has_sequential (a: ARRAY [T]; key: T): INTEGER
2    require
3      a.lower = 1  -- For convenience only, may be removed (see text).
4    local
5      i: INTEGER
6    do
7      from
8        i := 0 ; Result := 0
9      invariant
10       0 ≤ i ≤ a.count
11       Result ∈ [0, i]
12       Result ≠ 0  ⟹  key = a [Result]
13       Result = 0  ⟹  key ∉ elements (a [1..i])
14     until
15       i = a.upper
16     loop
17       i := i + 1
18       if a [i] = key then Result := i end
19     variant
20       a.upper − i + 1
21     end
22   ensure
23       Result ∈ [0, a.upper]
24       Result ≠ 0  ⟹  key = a [Result]
25       Result = 0  ⟹  key ∉ elements (a)
26 end
```

Figure 5: Search in an unsorted array.

> Although the basic idea of binary search is comparatively straightforward, the details can be surprisingly tricky, and many programmers have done it wrong the first few times they tried.

Reasoning carefully on the specification (at the domain-theory level) and the resulting invariant helps avoid mistakes.

For the present discussion it is interesting that the postcondition is the same as for sequential search (Section 4.1.3), so that we can see where the generalization strategy differs, taking advantage of the extra property that the array is sorted.

The algorithm and implementation now have the precondition

$$sorted\ (a),$$

where the domain-theory predicate $sorted\ (a)$, defined as

$$\forall j \in [\,a.lower\,..\,a.upper - 1] : a\ [j] \leq a\ [j{+}1],$$

expresses that an array is sorted upwards. The domain theorem on which binary search rests is that, for any value mid in $[\,i\,..\,j\,]$ (where i and j are valid indexes

of the array), and any value *key* of type T (the type of the array elements):

$$key \in elements(a[\,i\,..\,j\,]) \Longleftrightarrow \left(\begin{array}{c} key \leq a[mid] \wedge key \in elements(a[\,i\,..\,mid]) \\ \vee \\ key > a[mid] \wedge key \in elements(a[mid{+}1..j]) \end{array} \right). \tag{13}$$

This property leads to the key insight behind binary search, whose invariant follows from the postcondition by variable introduction, *mid* serving as that variable.

Formula (13) is not symmetric with respect to i and j; a symmetric version is possible, using in the second disjunct, "$\geq$" rather than "$>$" and *mid* rather than $mid + 1$. The form given in (13) has the advantage of using two mutually exclusive conditions in the comparison of *key* to $a\,[mid]$. As a consequence, we can limit ourselves to a value *mid* chosen in $[\,i\,..\,j\,-\,1]$ (rather than $[\,i\,..\,j]$) since the first disjunct does not involve j and the second disjunct cannot hold for $mid = j$ (the slice $a\,[mid + 1..j]$ being then empty). All these observations and choices have direct consequences on the program text, but are better handled at the specification (theory) level.

We will start for simplicity with the version (10) of the postcondition that only records presence or absence, repeated here for ease of reference:

$$\textbf{Result} \neq 0 \quad \Longleftrightarrow \quad key \in elements(a). \tag{14}$$

Duplicating the right-hand side of (14), writing a in slice form $a\,[1..\,a.upper]$, and applying constant relaxation twice, to the lower bound 1 and the upper bound $a.upper$, yields the essential invariant:

$$key \in elements(a[\,i\,..\,j\,]) \quad \Longleftrightarrow \quad key \in elements(a) \tag{15}$$

with the bounding invariant

$$1 \leq i \leq mid + 1 \quad \wedge \quad 1 \leq mid \leq j \leq a.upper \quad \wedge \quad i \leq j\,,$$

which combines the assumptions on *mid* necessary to apply (13)—also assumed in (15)—and the additional knowledge that $1 \leq i$ and $j \leq a.upper$.

The attraction of this presentation is that:

- The two clauses $key \leq a[mid]$ and $key > a[mid]$ of (13) are easy-to-test complementary conditions, suggesting a loop body that preserves the invariant by testing *key* against $a\,[mid]$ and going left or right as a result of the test.
- When $i = j$—the case that serves as exit condition—the left side of the equivalence (15) reduces to $key = a\,[i\,]$; evaluating this expression tells us whether *key* appeared at all in the entire array, the information we seek. In addition, we can obtain the stronger postcondition, (11)–(12), which gives **Result** its precise value, by simply assigning i to **Result**.

```
1  has_binary (a: ARRAY [T]; key: T): INTEGER
2    require
3      a.lower = 1  -- For convenience, see comment about has_sequential.
4      a.count > 0
5      sorted (a)
6    local
7      i, j, mid: INTEGER
8    do
9      from
10       i := 1; j := a.upper; mid := 1; Result := 0
11     invariant
12       1 ≤ i ≤ mid + 1 ∧ 1 ≤ mid ≤ j ≤ a.upper ∧ i ≤ j
13       key ∈ elements (a[i..j]) ⟺ key ∈ elements (a)
14     until
15       i = j
16     loop
17       mid := "A value in [i..j − 1]" -- In practice chosen as i + (j − i)//2
18       if a [mid] < key then i := mid + 1 else j := mid end
19     variant
20       j − i
21     end
22     if a [i] = key then Result := i end
23     ensure
24       0 ≤ Result ≤ n
25       Result ≠ 0 ⟹ key = a [Result]
26       Result = 0 ⟹ key ∉ elements (a)
27     end
```

Figure 6: Binary search.

This leads to the implementation in Figure 6.

To prove initiation, we note that initially *mid* is 1, so that $mid \in [i..j]$ is true. Consecution follows directly from (13).

For the expression assigned to *mid* in the loop, given in pseudocode as "A value in $[\, i \,..\, j - 1]$", the implementation indeed chooses, for efficiency, the midpoint of the interval $[\, i \,..\, j\,]$, which may be written $i + (j - i) \;//\; 2$ where "//" denotes integer division. In an implementation, this form is to be preferred to the simpler $(i + j) \;//\; 2$, whose evaluation on a computer may produce an integer overflow even when i, j, and their midpoint are all correctly representable on the computer's number system, but (because they are large) the sum $i + j$ is not [4]. In such a case the evaluation of $j - i$ is instead safe.

4.2 Arithmetic algorithms

Efficient implementations of the elementary arithmetic operations known since grade school require non-trivial algorithmic skills and feature interesting invariants, as the examples in this section demonstrate.

4.2.1 Integer division

The algorithm for integer division by successive differences computes the integer quotient q and the remainder r of two integers m and n. The postcondition reads

$$0 \leq r < m$$
$$n = m \cdot q + r.$$

The loop invariant consists of a bounding clause and an essential clause. The latter is simply an element of the postcondition:

$$n = m \cdot q + r.$$

The bounding clause weakens the other postcondition clause by keeping only its first part:

$$0 \leq r,$$

so that the dropped condition $r < m$ becomes the exit condition. As a consequence, $r \geq m$ holds in the loop body, and the assignment $r := r - m$ maintains the invariant property $0 \leq r$. It is straightforward to prove the implementation in Figure 7 correct with respect to this specification.

```
1  divided_diff  (n, m: INTEGER): (q, r: INTEGER)
2   require
3     n ≥ 0
4     m >0
5   do
6     from
7        r := n; q := 0
8     invariant
9        0 ≤ r
10       n = m · q + r
11    until
12       r < m
13    loop
14       r := r − m
15       q := q + 1
16    variant r
17    end
18  ensure
19    0 ≤ r < m
20    n = m · q + r
21  end
```

Figure 7: Integer division.

4.2.2 Greatest common divisor (with division)

Euclid's algorithm for the greatest common divisor offers another example where clearly separating between the underlying mathematical theory and the implementation yields a concise and convincing correctness argument. Sections 1.3 and 2 previewed this example by using the form that repeatedly subtracts one of the values from the other; here we will use the version that uses division.

The greatest common divisor $\gcd(a, b)$ is the greatest integer that divides both a and b, defined by the following axioms, where a and b are nonnegative integers such that at least one of them is positive ("\\" denotes integer remainder):

$$a \backslash\backslash \gcd(a, b) = 0$$
$$b \backslash\backslash \gcd(a, b) = 0$$
$$\forall d \in \mathbb{N} : (a \backslash\backslash d = 0) \wedge (b \backslash\backslash d = 0) \implies d \leq \gcd(a, b).$$

From this definition follow several properties of the gcd function:

Commutativity: $\gcd(a, b) = \gcd(b, a)$

Zero divisor: $\gcd(a, 0) = a$

Reduction: for $b > 0$, $\gcd(a, b) = \gcd(a \backslash\backslash b, b)$

The following property of the remainder operation is also useful:

Nonnegativity: for integers $a \geq 0$ and $b > 0$: $a \backslash\backslash b \geq 0$

From the obvious postcondition **Result** $= \gcd(a,\ b)$, we obtain the essential invariant in three steps:

1. By backward reasoning, derive the loop's postcondition $x = \gcd(a, b)$ from the routine's postcondition **Result** $= \gcd(a,b)$.
2. Using the *zero divisor* property, rewrite it as $\gcd(x, 0) = \gcd(a, b)$.
3. Apply constant relaxation, introducing variable y to replace 0.

This gives the essential invariant $\gcd(x, y) = \gcd(a, b)$ together with the bounding invariants $x > 0$ and $y \geq 0$. The corresponding implementation is shown in Figure 8.[5]

Initiation is established trivially. Consecution follows from the *reduction* property. Note that, unlike in the difference version (Section 1.3), we can arbitrarily decide always to divide x by y, rather than having to find out which of the two numbers is greater; hence the commutativity of gcd is not used in this proof.

[5]The variant is simply y, which is guaranteed to decrease at every iteration and can be bounded from below by the property $0 \leq x \backslash\backslash y < y$.

```
1  gcd_Euclid_division (a, b: INTEGER): INTEGER
2    require
3      a > 0
4      b ≥ 0
5    local
6      t, x, y: INTEGER
7    do
8      from
9        x := a
10       y := b
11     invariant
12       x > 0
13       y ≥ 0
14       gcd (x, y) = gcd (a, b)
15     until
16       y = 0
17     loop
18       t := y
19       y := x \\ y
20       x := t
21     variant y end
22     Result := x
23   ensure
24     Result = gcd (a, b)
25   end
```

Figure 8: Greatest common divisor with division.

4.2.3 Exponentiation by successive squaring

Suppose we do not have a built-in power operator and wish to compute m^n. We may of course multiply m by itself $n-1$ times, but a more efficient algorithm squares m for all 1s values in the binary representation of n. In practice, there is no need to compute this binary representation.

Given the postcondition

$$\textbf{Result} = m^n,$$

we first rewrite it into the obviously equivalent form $\textbf{Result} \cdot 1^1 = m^n$. Then, the invariant is obtained by double constant relaxation: the essential property

$$\textbf{Result} \cdot x^y = m^n$$

is easy to obtain initially (by setting **Result**, x, and y to 1, m, and n), yields the postcondition when $y = 0$, and can be maintained while progressing towards this situation thanks to the domain-theory properties

$$x^{2z} = (x^2)^{2z/2} \tag{16}$$

$$x^z = x \cdot x^{z-1}. \tag{17}$$

```
1  power_binary (m, n: INTEGER): INTEGER
2    require
3      n ≥ 0
4    local
5      x, y: INTEGER
6    do
7      from
8        Result := 1
9        x := m
10       y := n
11     invariant
12       y ≥ 0
13       Result · x^y = m^n
14     until y = 0
15     loop
16       if y.is_even then
17         x := x * x
18         y := y // 2
19       else
20         Result := Result * x
21         y := y − 1
22       end
23     variant y
24     end
25   ensure
26     Result = m^n
27   end
```

Figure 9: Exponentiation by successive squaring.

Using only (17) would lead to the inefficient $(n-1)$-multiplication algorithm, but we may use (16) for even values of $y = 2z$. This leads to the algorithm in Figure 9.

Proving initiation is trivial. Consecution is a direct application of the (16) and (17) properties.

4.2.4 Long integer addition

The algorithm for long integer addition computes the sum of two integers a and b given in any base as arrays of positional digits starting from the least significant position. For example, the array sequence $\langle 3, 2, 0, 1\rangle$ represents the number 138 in base 5 as $3 \cdot 5^0 + 2 \cdot 5^1 + 0 \cdot 5^2 + 1 \cdot 5^3 = 138$. For simplicity of representation, in this algorithm we use arrays indexed by 0, so that we can readily express the value encoded in base b by an array a as the sum:

$$\sum_{k=0}^{a.count} a[k] \cdot b^k.$$

The postcondition of the long integer addition algorithm has two clauses. One specifies that the pairwise sum of elements in a and b encodes the same number as **Result**:

$$\sum_{k=0}^{n-1}(a[k]+b[k])\cdot base^k = \sum_{k=0}^{n}\mathbf{Result}[k]\cdot base^k. \tag{18}$$

Result may have one more digit than a or b; hence the different bound in the two sums, where n denotes a's and b's length (normally written $a.count$ and $b.count$). The second postcondition clause is the consistency constraint that **Result** is indeed a representation in base *base*:

$$has\_base\ (\mathbf{Result},\ base), \tag{19}$$

where the predicate *has_base* is defined by a quantification over the array's length:

$$has\_base\ (v,\ b) \iff \forall k \in \mathbb{N} : 0 \le k < v.count \implies 0 \le v[k] < b.$$

Both postcondition clauses appear mutated in the loop invariant. First, we rewrite **Result** in slice form **Result** $[0..n]$ in (18) and (19). The first essential invariant clause follows by applying constant relaxation to (19), with the variable expression $i-1$ replacing constant n:

$$has\_base\ (\mathbf{Result}\ [0..i\ -\ 1],\ base).$$

The decrement is required because the loop updates i at the end of each iteration; it is a form of *aging* (see Section 3.2).

To get the other part of the essential invariant, we first highlight the last term in the summation on the right-hand side of (18):

$$\sum_{k=0}^{n-1}(a[k]+b[k])\cdot base^k = \mathbf{Result}[n]\cdot base^n + \sum_{k=0}^{n-1}\mathbf{Result}[k]\cdot base^k.$$

We then introduce variables i and *carry*, replacing constants n and **Result**$[n]$. Variable i is the loop counter, also mentioned in the other invariant clause; *carry*, as the name indicates, stores the remainder of each pairwise addition, which will be carried over to the next digit.

The domain property that the integer division by b of the sum of two b-base digits v_1, v_2 is less than b (all variables are integer):

$$b > 0\ \wedge\ v_1, v_2 \in [0..b-1] \implies (v_1+v_2)//b \in [0..b-1]$$

suggests the bounding invariant clause $0 \le carry < base$. Figure 10 shows the resulting implementation, where the most significant digit is set after the loop before terminating.

Initiation is trivial under the convention that an empty sum evaluates to zero. Consecution easily follows from the domain-theoretic properties of the operations in the loop body, and in particular from how the *carry* and the current digit d are set in each iteration.

```
addition (a, b: ARRAY [INTEGER];
          base: INTEGER): ARRAY [INTEGER]
  require
    base > 0
    a.count = b.count = n ≥ 1
    has_base (a, base)  -- a is a valid encoding in base base
    has_base (b, base)  -- b is a valid encoding in base base
    a.lower = b.lower = 0 -- For simplicity of representation
  local
    i, d, carry: INTEGER
  do
    Result := {0}^(n+1) -- Initialize Result to an array of size n + 1 with all 0s
    carry := 0
    from
      i := 0
    invariant
      ∑_{k=0}^{i-1} (a[k] + b[k])·base^k = carry·base^i + ∑_{k=0}^{i-1} Result[k]·base^k
      has_base (Result [0..i−1], base)
      0 ≤ carry < base
    until
      i = n
    loop
      d := a [i] + b [i] + carry
      Result [i] := d \\ base
      carry := d // base
      i := i + 1
    variant n − i end
    Result [n] := carry
  ensure
    ∑_{k=0}^{n-1} (a[k] + b[k])·base^k = ∑_{k=0}^{n} Result[k]·base^k
    has_base (Result, base)
  end
```

Figure 10: Long integer addition.

4.3 Sorting

A number of important algorithms sort an array based on pairwise comparisons and swaps of elements.The following domain-theory notations will be useful for arrays a and b:

- *perm* (a,b) expresses that the arrays are permutations of each other (their elements are the same, each occurring the same number of times as in the other array).
- *sorted* (a) expresses that the array elements appear in increasing order: $\forall i \in [a.lower..a.upper - 1]$: $a\,[i] \leq a\,[i + 1]$.

The sorting algorithms considered sort an array in place, with the specification:

$$
\begin{array}{l}
\mathit{sort}\ (a\colon\ \mathit{ARRAY}\,[T]) \\
\quad \textbf{require} \\
\qquad a.\mathit{lower} = 1 \\
\qquad a.\mathit{count} = n \geq 1 \\
\quad \textbf{ensure} \\
\qquad \mathit{perm}\ (a,\ \textbf{old}\ a) \\
\qquad \mathit{sorted}\ (a)
\end{array}
$$

The type T indicates a generic type that is totally ordered and provides the comparison operators $<$, $\leq$, $\geq$, and $>$. The precondition that the array be indexed from 1 and non-empty is a simplification that can be easily dropped; we adopt it in this section as it focuses the presentation of the algorithms on the interesting cases. For brevity, we also use n as an alias of a's length $a.\mathit{count}$.

The notation $a[\,i\,..\,j\,]{\sim}x$, for an array slice $a\ [\,i\,..\,j\,]$, a scalar value x, and a comparison operator $\sim$ among $<$, $\leq$, $\geq$, and $>$, denotes that all elements in the slice satisfy $\sim$ with respect to x: it is a shorthand for $\forall k \in [i..j]\colon a[k] \sim x$.

4.3.1 Quick sort: partitioning

At the core of the well-known quick sort algorithm lies the partitioning procedure, which includes loops with an interesting invariant; we analyze it in this section.

The procedure rearranges the elements in an array a according to an arbitrary value *pivot* given as input: all elements in positions up to **Result** included are no larger than *pivot*, and all elements in the other "high" portion (after position **Result**) of the array are no smaller than *pivot*. Formally, the postcondition is:

$$
\begin{array}{l}
0 \leq \textbf{Result} \leq n \\
\mathit{perm}\ (a,\ \textbf{old}\ a) \\
a\ \ [1..\,\textbf{Result}] \leq \mathit{pivot} \\
a\ [\textbf{Result} + 1..n] \geq \mathit{pivot}\,.
\end{array}
$$

In the special case where all elements in a are greater than or equal to *pivot*, **Result** will be zero, corresponding to the "low" portion of the array being empty.

Quick sort works by partitioning an array, and then recursively partitioning each of the two portions of the partition. The choice of *pivot* at every recursive call is crucial to guarantee a good performance of quick sort. Its correctness, however, relies solely on the correctness of *partition*, not on the choice of *pivot*. Hence the focus of this section is on *partition* alone.

The bulk of the loop invariant follows from the last three clauses of the postcondition. $\mathit{perm}\ (a,\ \textbf{old}\ a)$ appears unchanged in the essential invariant, denoting the fact that the whole algorithm does not change a's elements but

only rearranges them. The clauses comparing a's slices to *pivot* determine the rest of the essential invariant, once we modify them by introducing loop variables *low* and *high* decoupling and relaxing "constant" **Result**:

$$\begin{aligned} &\mathit{perm}\ (a, \textbf{old}\ a) \\ &a\ [1..\mathit{low} - 1] \leq \mathit{pivot} \\ &a\ [\mathit{high} + 1..n] \geq \mathit{pivot}\,. \end{aligned}$$

The formula $\mathit{low} = \mathit{high}$—removed when decoupling—becomes the main loop's exit condition. Finally, a similar variable introduction applied twice to the postcondition $0 \leq \textbf{Result} \leq n$ suggests the bounding invariant clause

$$1 \leq \mathit{low} \leq \mathit{high} \leq n\,.$$

The slice comparison $a\ [1..\mathit{low} - 1] \leq \mathit{pivot}$ also includes aging of variable *low*. This makes the invariant clauses fully symmetric, and suggests a matching implementation with two inner loops nested inside an overall outer loop. The outer loop starts with $\mathit{low} = 1$ and $\mathit{high} = n$ and terminates, with $\mathit{low} = \mathit{high}$, when the whole array has been processed. The first inner loop increments *low* until it points to an element that is larger than *pivot*, and hence is in the wrong portion of the array. Symmetrically, the outer loop decrements *high* until it points to an element smaller than *pivot*. After *low* and *high* are set by the inner loops, the outer loop swaps the corresponding elements, thus making progress towards partitioning the array. Figure 11 shows the resulting implementation. The closing conditional in the main routine's body ensures that **Result** points to an element no greater than *pivot*; this is not enforced by the loop, whose invariant leaves the value of $a\ [\mathit{low}]$ unconstrained. In particular, in the special case of all elements being no less than *pivot*, *low* and **Result** are set to zero after the loop.

In the correctness proof, it is useful to discuss the cases $a\ [\mathit{low}] < \mathit{pivot}$ and $a\ [\mathit{low}] \geq \mathit{pivot}$ separately when proving consecution. In the former case, we combine $a\ [1..\mathit{low} - 1] \leq \mathit{pivot}$ and $a\ [\mathit{low}] < \mathit{pivot}$ to establish the backward substitution $a\ [1..\mathit{low}] \leq \mathit{pivot}$. In the latter case, we combine $\mathit{low} = \mathit{high}$, $a\ [\mathit{high} + 1..n] \geq \mathit{pivot}$ and $a\ [\mathit{low}] \geq \mathit{pivot}$ to establish the backward substitution $a\ [\mathit{low}..n] \geq \mathit{pivot}$. The other details of the proof are straightforward.

4.3.2 Selection sort

Selection sort is a straightforward sorting algorithm based on a simple idea: to sort an array, find the smallest element, put it in the first position, and repeat recursively from the second position on. Pre- and postcondition are the usual ones for sorting (see Section 4.3), and hence require no further comment.

The first postcondition clause $\mathit{perm}\ (a, \textbf{old}\ a)$ is also an essential loop invariant:

$$\mathit{perm}\ (a, \textbf{old}\ a). \tag{20}$$

If we introduce a variable i to iterate over the array, another essential invariant clause is derived by writing a in slice form $a\,[1..\,n]$ and then by relaxing n into i:

$$sorted\ (a\ \ [1..\,i\,]) \tag{21}$$

with the bounding clause

$$1 \leq i \leq n, \tag{22}$$

```
1  partition (a: ARRAY [T]; pivot: T): INTEGER
2    require
3      a.lower = 1
4      a.count = n ≥ 1
5    local
6      low, high: INTEGER
7    do
8      from low := 1 ; high := n
9      invariant
10       1 ≤ low ≤ high ≤ n
11       perm (a, old a)
12       a [1.. low − 1] ≤ pivot
13       a [high + 1..n] ≥ pivot
14     until low = high
15     loop
16       from −− This loop increases low
17       invariant −− Same as outer loop
18       until low = high ∨ a[low] > pivot
19       loop low := low + 1 end
20       from −− This loop decreases high
21       invariant −− Same as outer loop
22       until low = high ∨ a[high] < pivot
23       loop high := high − 1 end
24       a.swap (low, high) −− Swap the elements in positions low and high
25     variant high − low end
26     if a [low] ≥ pivot then
27       low := low − 1
28       high := low
29     end
30     Result := low
31   ensure
32     0 ≤ Result ≤ n
33     perm (a, old a)
34     a [1.. Result] ≤ pivot
35     a [Result + 1..n] ≥ pivot
36   end
```

Figure 11: Quick sort: partitioning.

which ensures that the sorted slice $a\,[1..\ i]$ is always non-empty. The final component of the invariant is also an essential weakening of the postcondition, but is less straightforward to derive by syntactic mutation. If we split $a\ \ [1..\ n]$ into the concatenation $a\ \ [1..\ i\ -\ 1] \circ a\ \ [\,i\,..\,n]$, we notice that *sorted* $(a\ \ [1..\ i\ -\ 1] \circ a\ \ [\,i\,..\,n])$ implies

$$\forall k \in [i..n] : a\ \ [1..\ i\ -\ 1] \leq a[k] \tag{23}$$

as a special case. Formula (23) guarantees that the slice $a[\,i\,..\,n]$, which has not been sorted yet, contains elements that are no smaller than any of those in the sorted slice $a\,[1..\ i\ -\ 1]$.

The loop invariants (20)–(22) apply—possibly with minimal changes due to inessential details in the implementation—for any sorting algorithm that sorts an array sequentially, working its way from lower to upper indices. To implement the behavior specific to selection sort, we introduce an inner loop that finds the minimum element in the slice $a\ \ [\,i\,..\,n]$, which is not yet sorted. To this end, it uses variables j and m: j scans the slice sequentially starting from position $i+1$; m points to the minimum element found so far. Correspondingly, the inner loop's postcondition is $a[m] \leq a[\,i\,..\,n]$, which induces the essential invariant clause

$$a\ \ [m] \leq a\ \ [\,i\,..\,j\ -\ 1] \tag{24}$$

specific to the inner loop, by constant relaxation and aging. The outer loop's invariant (23) clearly also applies to the inner loop—which does not change i or n—where it implies that the element in position m is an upper bound on all elements already sorted:

$$a\ \ [1..\ i\ -\ 1] \leq a\ \ [m]. \tag{25}$$

Also specific to the inner loop are more complex bounding invariants relating the values of i, j, and m to the array bounds:

$$1 \leq i < j \leq n + 1$$
$$i \leq m < j.$$

The implementation in Figure 12 follows these invariants. The outer loop's only task is then to swap the "minimum" element pointed to by m with the lowest available position pointed to by i.

The most interesting aspect of the correctness proof is proving consecution of the outer loop's invariant clause (21), and in particular that $a[\,i\,] \leq a[\,i\ +\ 1]$. To this end, notice that (24) guarantees that $a\ \ [m]$ is the minimum of all elements in positions from i to n; and (25) that it is an upper bound on the other elements in positions from 1 to $i-1$. In particular, $a[m] \leq a[\,i{+}1]$ and $a[\,i\ -\ 1] \leq a[m]$ hold before the swap on line 30. After the swap, $a[\,i\,]$ equals the previous value of $a[m]$, thus $a[\,i\ -\ 1] \leq a[\,i\,] \leq a[\,i\ +\ 1]$ holds as required. A similar reasoning proves the inductiveness of the main loop's other invariant clause (23).

```
1  selection_sort  (a: ARRAY [T])
2   require
3     a.lower = 1
4     a.count = n ≥ 1
5   local
6     i, j, m: INTEGER
7   do
8     from i := 1
9     invariant
10      1 ≤ i ≤ n
11      perm (a, old a)
12      sorted (a [1.. i])
13      ∀k ∈ [i..n]: a [1..i − 1] ≤ a [k]
14    until
15      i = n
16    loop
17      from j := i + 1 ; m := i
18      invariant
19        1 ≤ i < j ≤ n + 1
20        i ≤ m < j
21        perm (a, old a)
22        sorted (a [1.. i])
23        a [1.. i − 1] ≤ a [m] ≤ a [i .. j − 1]
24      until
25        j = n + 1
26      loop
27        if a [j] < a [m] then m := j end
28        j := j + 1
29      variant n − i − j end
30      a.swap (i, m)  -- Swap the elements in positions i and m
31      i := i + 1
32    variant n − i end
33  ensure
34    perm (a, old a)
35    sorted (a)
36  end
```

Figure 12: Selection sort.

4.3.3 Insertion sort

Insertion sort is another sub-optimal sorting algorithm that is, however, simple to present and implement, and reasonably efficient on arrays of small size. As the name suggests, insertion sort hinges on the idea of re-arranging elements in an array by inserting them in their correct positions with respect to the sorting order; insertion is done by shifting the elements to make room for insertion. Pre- and postcondition are the usual ones for sorting (see Section 4.3 and the comments in the previous subsections).

The main loop's essential invariant is as in selection sort (Section 4.3.2) and other similar algorithms, as it merely expresses the property that the sorting has progressed up to position i and has not changed the array content:

$$sorted\ (a\ [1..\ i\,]) \tag{26}$$

$$perm\ (a,\ \textbf{old}\ a). \tag{27}$$

This essential invariant goes together with the bounding clause $1 \leq i \leq n$.

The main loop includes an inner loop, whose invariant captures the specific strategy of insertion sort. The outer loop's invariant (27) must be weakened, because the inner loop overwrites $a\ [\,i\,]$ while progressively shifting to the right elements in the slice $a\ [1..\ j\,]$. If a local variable v stores the value of $a\ [\,i\,]$ before entering the inner loop, we can weaken (27) as:

$$perm\ (a\ [1..\ j\,]\ \circ\ v\ \circ\ a[j\ +\ 2..n],\ \textbf{old}\ a), \tag{28}$$

where "$\circ$" is the concatenation operator; that is, a's element at position $j+1$ is the current candidate for inserting v—the value temporarily removed. After the inner loop terminates, the outer loop will put v back into the array at position $j + 1$ (line 28 in Figure 13), thus restoring the stronger invariant (27) (and establishing inductiveness for it).

The clause (26), crucial for the correctness argument, is also weakened in the inner loop. First, we "age" i by replacing it with $i - 1$; this corresponds to the fact that the outer loop increments i at the beginning, and will then re-establish (26) only at the *end* of each iteration. Therefore, the inner loop can only assume the weaker invariant:

$$sorted\ (a\ [1..\ i\ -\ 1]) \tag{29}$$

that is not invalidated by shifting (which only temporarily duplicates elements). Shifting has, however, another effect: since the slice $a[j\ +\ 1..\,i]$ contains elements shifted up from the sorted portion, the slice $a[j\ +\ 1..\,i]$ is itself sorted, thus the essential invariant:

$$sorted\ (a\ [j\ +\ 1..\,i]). \tag{30}$$

We can derive the pair of invariants (29)–(30) from the inner loop's postcondition (26): write $a\ [1..\ i]$ as $a\ [1..\ i\ -\ 1]\ \circ\ a[\,i\,..\,i]$; weaken the formula $sorted\ (a\ [1..\ i\ -\ 1]\ \circ\ a[\,i\,..\,i])$ into the conjunction of $sorted(\ a\ [1..\ i\ -\ 1])$ and $sorted\ (a[\,i\,..\,i])$; replace one occurrence of constant i in the second conjunct by a fresh variable j and age to derive $sorted\ (a\ [j\ +\ 1..\,i])$.

Finally, there is another essential invariant, specific to the inner loop. Since the loop's goal is to find a position, pointed to by $j + 1$, before i where v can be inserted, its postcondition is:

$$v \leq a\ [j\ +\ 1..\,i], \tag{31}$$

which is also a suitable loop invariant, combined with a bounding clause that constrains j and i:

$$0 \leq j\ < i \leq n. \tag{32}$$

Overall, clauses (28)–(32) are the inner loop invariant; and Figure 13 shows the matching implementation.

```
1  insertion_sort  (A: ARRAY [T])
2    require
3      a.lower = 1  ;  a.count = n ≥ 1
4    local
5      i, j: INTEGER ; v : T
6    do
7      from i := 1
8      invariant
9        1 ≤ i ≤ n
10       perm (a, old a)
11       sorted (a [1.. i])
12     until i = n
13     loop
14       i := i + 1
15       v := a [i]
16       from j := i − 1
17       invariant
18         0 ≤ j < i ≤ n
19         perm (a [1.. j] ∘ v ∘ a[j + 2..n], old a)
20         sorted (a [1.. i − 1])
21         sorted (a [j + 1..i])
22         v ≤ a [j + 1..i]
23       until j = 0 or a [j] ≤ v
24       loop
25         a [j + 1] := a [j]
26         j := j − 1
27       variant j − i end
28       a [j + 1] := v
29     variant n − i end
30   ensure
31     perm (a, old a)
32     sorted (a)
33   end
```

Figure 13: Insertion sort.

As usual for this kind of algorithms, the crux of the correctness argument is proving that the outer loop's essential invariant is inductive, based on the inner loop's. The formal proof uses the following informal argument. Formulas (29) and (31) imply that inserting v at $j + 1$ does not break the sortedness of the slice $a\ [1..\ j + 1]$. Furthermore, (30) guarantees that the elements in the "upper" slice $a\ [j + 1..\ i]$ are also sorted with $a\ [j] \leq a[j + 1] \leq a[j + 2]$. (The detailed argument would discuss the cases $j = 0$, $0 < j < i - 1$, and $j = i - 1$.) In all, the whole slice $a\ [1..\ i]$ is sorted, as required by (26).

```
1  bubble_sort_basic  (a: ARRAY [T])
2   require
3     a.lower = 1  ;  a.count = n ≥ 1
4   local
5     swapped: BOOLEAN
6     i: INTEGER
7   do
8     from swapped := True
9     invariant
10      perm (a, old a)
11      ¬ swapped ⟹ sorted (a)
12    until ¬ swapped
13    loop
14      swapped := False
15      from i := 1
16      invariant
17        1 ≤ i ≤ n
18        perm (a, old a)
19        ¬ swapped ⟹ sorted (a [1.. i])
20      until i = n
21      loop
22        if a [i] > a [i + 1] then
23          a.swap (i, i + 1)-- Swap the elements in positions i and i + 1
24          swapped := True
25        end
26        i := i + 1
27      variant n − i end
28    variant |inversions (a)|
29    end
30  ensure
31    perm (a, old a)
32    sorted (a)
33  end
```

Figure 14: Bubble sort (basic version).

4.3.4 Bubble sort (basic)

As a sorting method, bubble sort is known not for its performance but for its simplicity [37, Vol. 3, Sec. 5.2.2]. It relies on the notion of *inversion*: a pair of elements that are not ordered, that is, such that the first is greater than the second. The straightforward observation that an array is sorted if and only if it has no inversions suggests to sort an array by iteratively removing all inversions. Let us present invariants that match such a high-level strategy, deriving them from the postcondition (which is the same as the other sorting algorithms of this section).

The postcondition *perm* (*a*, **old** *a*) that *a*'s elements be not changed is also

an invariant of the two nested loops used in bubble sort. The other postcondition *sorted* (a) is instead weakened, but in a way different than in other sorting algorithms seen before. We introduce a Boolean flag *swapped*, which records if there is *some* inversion that has been removed by swapping a pair of elements. When *swapped* is false after a complete scan of the array a, no inversions have been found, and hence a is sorted. Therefore, we use $\neg$ *swapped* as exit condition of the main loop, and the weakened postcondition

$$\neg \mathit{swapped} \Longrightarrow \mathit{sorted}\ (a) \tag{33}$$

as its essential loop invariant.

The inner loop performs a scan of the input array that compares all pairs of adjacent elements and swaps them when they are inverted. Since the scan proceeds linearly from the first element to the last one, we get an essential invariant for the inner loop by replacing n by i in (33) written in slice form:

$$\neg \mathit{swapped} \Longrightarrow \mathit{sorted}\ (a\ [1..\ i])\,. \tag{34}$$

The usual bounding invariant $1 \leq i \leq n$ and the outer loop's invariant clause *perm* $(a,$ **old** $a)$ complete the inner loop invariant.

The implementation is now straightforward to write as in Figure 14. The inner loop, in particular, sets *swapped* to **True** whenever it finds some inversion while scanning. This signals that more scans are needed before the array is certainly sorted.

Verifying the correctness of the annotated program in Figure 14 is easy, because the essential loop invariants (33) and (34) are trivially true in all iterations where *swapped* is set to **True**. On the other hand, this style of specification makes the termination argument more involved: the outer loop's variant (line 28 in Figure 14) must explicitly refer to the number of inversions left in a, which are decreased by complete executions of the inner loop.

4.3.5 Bubble sort (improved)

The inner loop in the basic version of bubble sort—presented in Section 4.3.4—always performs a complete scan of the n-element array a. This is often redundant, because swapping adjacent inverted elements guarantees that the largest misplaced element is sorted after each iteration. Namely, the largest element reaches the rightmost position after the first iteration, the second-largest one reaches the penultimate position after the second iteration, and so on. This section describes an implementation of bubble sort that takes advantage of this observation to improve the running time.

The improved version still uses two nested loops. The outer loop's essential invariant has two clauses:

$$\mathit{sorted}\ (a\ [\,i\,..\,n]) \tag{35}$$

is a weakening of the postcondition that encodes the knowledge that the "upper" part of array a is sorted, and

$$i < n \implies a\,[1..\ i] \leq a[\,i\ +\ 1] \tag{36}$$

specifies that the elements in the unsorted slice $a\,[1..\,i]$ are no larger than the first "sorted" element $a[\,i\,+\,1]$. The expression $a\,[1..\,i] \leq a[\,i\,+\,1]$ is a mutation (constant relaxation and aging) of $a\,[1..\,n] \leq a[n]$, which is, in turn, a domain property following from the postcondition. Variable i is now used in the outer loop to mark the portion still to be sorted; correspondingly, (36) is well-defined only when $i < n$, and the bounding invariant clause $1 \leq i \leq n$ is also part of the outer loop's specification.

```
1  bubble_sort_improved (a: ARRAY [T])
2    require
3      a.lower = 1  ;  a.count = n ≥ 1
4    local
5      i, j: INTEGER
6    do
7      from i := n
8      invariant
9        1 ≤ i ≤ n
10       perm (a, old a)
11       sorted (a [i .. n])
12       i < n ⟹ a[1..i] ≤ a[i + 1]
13     until i = 1
14     loop
15       from j := 1
16       invariant
17         1 ≤ i ≤ n
18         1 ≤ j ≤ i
19         perm (a, old a)
20         sorted (a [i .. n])
21         i < n ⟹ a[1..i] ≤ a[i + 1]
22         a [1.. j] ≤ a[j]
23       until j = i
24       loop
25         if a [j] > a [j + 1] then a.swap (j, j + 1) end
26         j := j + 1
27       variant i − j
28       end
29       i := i − 1
30     variant i
31     end
32   ensure
33     perm (a, old a)
34     sorted (a)
35   end
```

Figure 15: Bubble sort (improved version).

Continuing with the same logic, the inner loop's postcondition:

$$a\ [1..\ i] \leq a[\,i\,] \tag{37}$$

states that the largest element in the slice $a\ [1..\ i]$ has been moved to the highest position. Constant relaxation, replacing i (not changed by the inner loop) with a fresh variable j, yields a new essential component of the inner loop's invariant:

$$a\ [1..\ j] \leq a[j]. \tag{38}$$

The outer loop's invariant and the bounding clause $1 \leq j \leq i$ complete the specification of the inner loop. Figure 15 displays the corresponding implementation.

The correctness proof follows standard strategies. In particular, the inner loop's postcondition (37)—i.e., the inner loop's invariant when $j = i$—implies $a[\,i\ -\ 1] \leq a[\,i\,]$ as a special case. This fact combines with the other clause (36) to establish the inductiveness of the main loop's essential clause:

$$\mathit{sorted}\ (a[\,i\,..\,n]).$$

Finally, proving termination is trivial for this program because each loop has an associated iteration variable that is unconditionally incremented or decremented.

4.3.6 Comb sort

In an attempt to improve performance in critical cases, comb sort generalizes bubble sort based on the observation that small elements initially stored in the right-most portion of an array require a large number of iterations to be sorted. This happens because bubble sort swaps adjacent elements; hence it takes n scans of an array of size n just to bring the smallest element from the right-most nth position to the first one, where it belongs. Comb sort adds the flexibility of swapping non-adjacent elements, thus allowing for a faster movement of small elements from right to left. A sequence of non-adjacent equally-spaced elements also conveys the image of a comb's teeth, hence the name "comb sort".

Let us make this intuition rigorous and generalize the loop invariants, and the implementation, of the basic bubble sort algorithm described in Section 4.3.4. Comb sort is also based on swapping elements, therefore the—now well-known—invariant *perm* (a, **old** a) also applies to its two nested loops. To adapt the other loop invariant (33), we need a generalization of the predicate *sorted* that fits the behavior of comb sort. Predicate *gap_sorted* (a, d), defined as:

$$\mathit{gap\_sorted}(a, d) \iff \forall k \in [\,a.\mathit{lower}..\,a.\mathit{upper} - d] : a\ [k] \leq a\ [k + d]$$

holds for arrays a such that the subsequence of d-spaced elements is sorted. Notice that, for $d = 1$, *gap_sorted* reduces to *sorted*:

$$\mathit{gap\_sorted}\ (a,\ 1) \iff \mathit{sorted}\ (a).$$

This fact will be used to prove the postcondition from the loop invariant upon termination.

With this new piece of domain theory, we can easily generalize the essential and bounding invariants of Figure 14 to comb sort. The outer loop considers decreasing gaps; if variable *gap* stores the current value, the bounding invariant

$$1 \leq gap \leq n$$

defines its variability range. Precisely, the main loop starts with with $gap = n$ and terminates with $gap = 1$, satisfying the essential invariant:

$$\neg\, swapped \Longrightarrow gap\_sorted\ (a,\ gap). \tag{39}$$

The correctness of comb sort does not depend on how *gap* is decreased, as long as it eventually reaches 1; if *gap* is initialized to 1, comb sort behaves exactly as bubble sort. In practice, it is customary to divide *gap* by some chosen parameter c at every iteration of the main loop.

Let us now consider the inner loop, which compares and swaps the subsequence of d-spaced elements. The bubble sort invariant (34) generalizes to:

$$\neg\, swapped \Longrightarrow gap\_sorted\ (a\ [1..\ i\ -\ 1\ +\ gap],\ gap) \tag{40}$$

and its matching bounding invariant is:

$$1 \leq i < i + gap \leq n + 1$$

so that when $i = n + 1 + gap$ the inner loop terminates and (40) is equivalent to (39). This invariant follows from constant relaxation and aging; the substituted expression $i\ -\ 1\ +\ gap$ is more involved, to accommodate how i is used and updated in the inner loop, but is otherwise semantically straightforward.

The complete implementation is shown in Figure 16. The correctness argument is exactly as for bubble sort in Section 4.3.4, but exploits the properties of the generalized predicate *gap_sorted* instead of the simpler *sorted*.

4.4 Dynamic programming

Dynamic programming is an algorithmic technique used to compute functions that have a natural recursive definition. Dynamic programming algorithms construct solutions iteratively and store the intermediate results, so that the solution to larger instances can reuse the previously computed solutions for smaller instances. This section presents a few examples of problems that lend themselves to dynamic programming solutions.

4.4.1 Unbounded knapsack problem with integer weights

We have an unlimited collection of items of n different types. An item of type k, for $k = 1, \ldots, n$, has weight $w[k]$ and value $v[k]$. The unbounded knapsack

```
1  comb_sort (a: ARRAY [T])
2    require
3      a.lower = 1  ;  a.count = n ≥ 1
4    local
5      swapped: BOOLEAN
6      i, gap: INTEGER
7    do
8      from swapped := True ; gap := n
9      invariant
10       1 ≤ gap ≤ n
11       perm (a, old a)
12       ¬ swapped ⟹ gap_sorted (a, gap)
13     until
14       ¬ swapped and gap = 1
15     loop
16       gap := max (1, gap // c)
17        -- c > 1 is a parameter whose value does not affect correctness
18       swapped := False
19       from i := 1
20       invariant
21         1 ≤ gap ≤ n
22         1 ≤ i < i + gap ≤ n + 1
23         perm (a, old a)
24         ¬ swapped ⟹ gap_sorted (a [1..i − 1 + gap], gap)
25       until
26         i + gap = n + 1
27       loop
28         if a [i] > a[i + gap] then
29           a.swap (i, i + gap)
30           swapped := True
31         end
32         i := i + 1
33       variant n + 1 − gap − i end
34     variant |inversions (a)| end
35   ensure
36     perm (a, old a)
37     sorted (a)
38   end
```

Figure 16: Comb sort.

problem asks what is the maximum overall value that one can carry in a knapsack whose weight limit is a given *weight*. The attribute "unbounded" refers to the fact that we can pick as many object of any type as we want: the only limit is given by the input value of *weight*, and by the constraint that we cannot store fractions of an item—either we pick it or we don't.

Any vector s of n nonnegative integers defines a selection of items, whose

overall weight is given by the scalar product:

$$s \cdot w \quad = \quad \sum_{1 \leq k \leq n} s[k]w[k]$$

and whose *overall value* is similarly given by the scalar product $s \cdot v$. Using this notation, we introduce the domain-theoretical function *max_knapsack* which defines the solution of the knapsack problem given a weight limit b and items of n types with weight and value given by the vectors w and v:

$$\mathit{max\_knapsack}(b, v, w, n) = \kappa \Longleftrightarrow \left(\begin{array}{rl} \exists s \in \mathbb{N}^n : & s \cdot w \leq b \ \wedge \ s \cdot v = \kappa \\ & \wedge \\ \forall t \in \mathbb{N}^n : & t \cdot w \leq b \implies t \cdot v \leq \kappa \end{array} \right),$$

that is, the largest value achievable with the given limit. Whenever weights w, values v, and number n of item types are clear from the context, we will abbreviate *max_knapsack* $(b,\ v,\ w,\ n)$ by just $K(b)$.

The unbounded knapsack problem is NP-complete [24, 36]. It is, however, *weakly* NP-complete [51], and in fact it has a nice solution with pseudo-polynomial complexity based on a recurrence relation, which suggests a straightforward dynamic programming algorithm. The recurrence relation defines the value of $K(b)$ based on the values $K(b')$ for $b' < b$.

The base case is for $b = 0$. If we assume, without loss of generality, that no item has null weight, it is clear that we cannot store anything in the knapsack without adding some weight, and hence the maximum value attainable with a weight limit zero is also zero: $K(0) = 0$. Let now b be a generic weight limit greater than zero. To determine the value of $K(b)$, we make a series of attempts as follows. First, we select some item of type k, such that $w[k] \leq b$. Then, we recursively consider the best selection of items for a weight limit of $b - w[k]$; and we set up a new selection by adding one item of type k to it. The new configuration has weight no greater than $b - w[k] + w[k] = b$ and value

$$v[k] + K(b - w[k]),$$

which is, by inductive hypothesis, the largest achievable by adding an element of type k. Correspondingly, the recurrence relation defines $K(b)$ as the maximum among all values achievable by adding an object of some type:

$$K(b) = \begin{cases} 0 & b = 0 \\ \max \left\{ v[k] + K(b - w[k]) \;\middle|\; k \in [1..n] \text{ and } 0 \leq w[k] \leq b \right\} & b > 0. \end{cases} \tag{41}$$

The dynamic programming solution to the knapsack problem presented in this section computes the recursive definition (41) for increasing values of b. It inputs arrays v and w (storing the values and weights of all elements), the number n of element types, and the weight bound *weight*. The precondition requires that *weight* be nonnegative, that all element weights w be positive, and

that the arrays v and w be indexed from 1 to n:

$$\begin{aligned} &\mathit{weight} \geq 0 \\ &w > 0 \\ &v.\mathit{lower} = w.\mathit{lower} = 1 \\ &v.\mathit{upper} = w.\mathit{upper} = n\,. \end{aligned}$$

The last two clauses are merely for notational convenience and could be dropped. The postcondition states that the routine returns the value $K(\mathit{weight})$ or, with more precise notation:

$$\textbf{Result} = \mathit{max\_knapsack}\ (\mathit{weight},\ v,\ w,\ n).$$

The main loop starts with $b = 0$ and continues with unit increments until $b = \mathit{weight}$; each iteration stores the value of $K(b)$ in the local array m, so that $m\ [\mathit{weight}]$ will contain the final result upon termination. Correspondingly, the main loop's essential invariant follows by constant relaxation:

- Variable $m\ [\mathit{weight}]$ replaces constant **Result**, thus connecting m to the returned result.
- The range of variables $[1..\ b]$ replaces constant *weight*, thus expressing the loop's incremental progress.

The loop invariant is thus:

$$\forall y \in [0..b] : \quad m[y] = \mathit{max\_knapsack}(y, v, w, n), \tag{42}$$

which goes together with the bounding clause

$$0 \leq b \leq \mathit{weight}$$

that qualifies b's variability domain. With a slight abuse of notation, we concisely write (42), and similar expressions, as:

$$m[0..b] = \mathit{max\_knapsack}([0..b], v, w, n). \tag{43}$$

The inner loop computes the maximum of definition (41) iteratively, for all element types j, where $1 \leq j \leq n$. To derive its essential invariant, we first consider its postcondition (similarly as the analysis of selection sort in Section 4.3.2). Since the inner loop terminates at the end of the outer loop's body, the inner's postcondition is the outer's invariant (43). Let us rewrite it by highlighting the value $m[b]$ computed in the latest iteration:

$$m[0..b-1] \ = \ \mathit{max\_knapsack}\ ([0..b-1], v, w, n) \tag{44}$$

$$m[b] \ = \ \mathit{best\_value}\ (b, v, w, n, n). \tag{45}$$

Function *best_value* is part of the domain theory for knapsack, and it expresses the "best" value that can be achieved given a weight limit of b, $j \leq n$ element

types, and assuming that the values $K(b')$ for lower weight limits $b' < b$ are known:

$$\mathit{best\_value}\ (b, v, w, j, n) = \max\Big\{v[k]{+}K(b{-}w[k])\ \Big|\ k \in [1..j] \text{ and } 0 \le w[k] \le b\Big\}.$$

If we substitute variable j for constant n in (45), expressing the fact that the inner loop tries one element type at a time, we get the inner loop essential invariant:

$$\begin{aligned} m[0..b-1] &= \mathit{max\_knapsack}\ ([0..b-1], v, w, n) \\ m[b] &= \mathit{best\_value}\ (b, v, w, j, m). \end{aligned}$$

The obvious bounding invariants $0 \le b \le \mathit{weight}$ and $0 \le j \le n$ complete the inner loop's specification. Figure 17 shows the corresponding implementation.

The correctness proof reverses the construction we highlighted following the usual patterns seen in this section. In particular, notice that:

- When $j = n$ the inner loop terminates, thus establishing (44) and (45).
- (44) and (45) imply (43) because the recursive definition (41) for some b only depends on the previous values for $b' < b$, and (44) guarantees that m stores those values.

4.4.2 Levenshtein distance

The *Levenshtein distance* of two sequences s and t is the minimum number of elementary edit operations (deletion, addition, or substitution of one element in either sequence) necessary to turn s into t. The distance has a natural recursive definition:

$$\mathit{distance}(s,t) = \begin{cases} 0 & m = n = 0 \\ m & m > 0, n = 0 \\ n & n > 0, m = 0 \\ \mathit{distance}\Big(\ s[1..m-1], t[1..n-1]\ \Big) & m > 0, n > 0, s[m] = t[n] \\ 1 + \min \left(\begin{array}{l} \mathit{distance}(s[1..m-1], t), \\ \mathit{distance}(s, t[1..n-1]), \\ \mathit{distance}(s[1..m-1], t[1..n-1]) \end{array} \right) & m > 0, n > 0, s[m] \neq t[n], \end{cases}$$

where m and n, respectively, denote s's and t's length (written $s.\mathit{count}$ and $t.\mathit{count}$ when s and t are arrays). The first three cases of the definition are trivial and correspond to when s, t, or both are empty: the only way to get a non-empty string from an empty one is by adding all the former's elements. If s and t are both non-empty and their last elements coincide, then the same number of operations as for the shorter sequences $s[1..m-1]$ and $t[1..n-1]$ (which omit the last elements) are needed. Finally, if s's and t's last elements differ, there are three options: (1) delete $s[m]$ and then edit $s[1..m-1]$ into t; (2) edit s into $t[1..n-1]$ and then add $t[n]$ to the result; (3) substitute $t[n]$ for $s[m]$ and then edit the rest $s[1..m-1]$ into $t[1..n-1]$. Whichever of the

```
1  knapsack (v, w: ARRAY [INTEGER]; n, weight: INTEGER): INTEGER
2    require
3      weight ≥ 0
4      w > 0
5      v.lower = w.lower = 1
6      v.upper = w.upper = n
7    local
8      b, j: INTEGER
9      m: ARRAY [INTEGER]
10   do
11     from b := 0 ; m [0] := 0
12     invariant
13       0 ≤ b ≤ weight
14       m [0..b] = max_knapsack ([0..b], v, w, n)
15     until b = weight
16     loop
17       b := b + 1
18       from j := 0 ; m [b] := m [b − 1]
19       invariant
20         0 ≤ b ≤ weight
21         0 ≤ j ≤ n
22         m [0..b − 1] = max_knapsack ([0..b − 1], v, w, n)
23         m [b] = best_value (b, v, w, j, n)
24       until j = n
25       loop
26         j := j + 1
27         if w [j] ≤ b and m [b] < v [j] + m [b − w [j]] then
28           m [b] := v [j] + m [b − w [j]]
29         end
30       variant n − j end
31     variant weight − b end
32     Result := m [weight]
33   ensure
34     Result = max_knapsack (weight, v, w, n)
35   end
```

Figure 17: Unbounded knapsack problem with integer weights.

options (1), (2), and (3) leads to the minimal number of edit operations is the Levenshtein distance.

It is natural to use a dynamic programming algorithm to compute the Levenshtein distance according to its recursive definition. The overall specification, implementation, and the corresponding proofs, are along the same lines as the knapsack problem of Section 4.4.1; therefore, we briefly present only the most important details. The postcondition is simply

$$\textbf{Result} = \mathit{distance}\,(s, t).$$

The implementation incrementally builds a bidimensional matrix d of distances such that the element $d[i, j]$ stores the Levenshtein distance of the sequences $s[1..i]$ and $t[1..j]$. Correspondingly, there are two nested loops: the outer loop iterates over rows of d, and the inner loop iterates over each column of d. Their essential invariants express, through quantification, the partial progress achieved after each iteration:

$$\forall h \in [0..i-1], \forall k \in [0..n] : d[h,k] = \mathit{distance}(s[1..h], t[1..k])$$
$$\forall h \in [0..i-1], \forall k \in [0..j-1] : d[h,k] = \mathit{distance}(s[1..h], t[1..k]).$$

The standard bounding invariants on the loop variables i and j complete the specification.

Figure 18 shows the implementation, which uses the compact **across** notation for loops, similar to "for" loops in other languages. The syntax

across $[a..b]$ **invariant** I **as** k **loop** B **end**

is simply a shorthand for:

from $k := a$ **invariant** I **until** $k = b + 1$ **loop** B ; $k := k + 1$ **end**

For brevity, Figure 18 omits the obvious loop invariants of the initialization loops at lines 11 and 12.

4.5 Computational geometry: Rotating calipers

The *diameter* of a polygon is its maximum width, that is the maximum distance between any pair of its points. For a convex polygon, it is clear that a pair of vertices determine the diameter (such as vertices p_3 and p_7 in Figure 19). Shamos showed [59] that it is not necessary to check all $O(n^2)$ pairs of vertices: his algorithm, described later, runs in time $O(n)$. The correctness of the algorithm rests on the notions of *lines of support* and antipodal points. A line of support is analogue to a tangent: a line of support of a convex polygon p is a line that intersects p such that the interior of p lies entirely to one side of the line. An antipodal pair is then any pair of p's vertices that lie on two *parallel* lines of support. It is a geometric property that an antipodal pair determines the diameter of any polygon p, and that a convex polygon with n vertices has $O(n)$ antipodal pairs. Figure 19(a), for example, shows two parallel lines of support that identify the antipodal pair (p_1, p_5).

Shamos's algorithms efficiently enumerates all antipodal pairs by rotating two lines of support while maintaining them parallel. After a complete rotation around the polygon, they have touched all antipodal pairs, and hence the algorithm can terminate. Observing that two parallel support lines resemble the two jaws of a caliper, Toussaint [62] suggested the name "rotating calipers" to describe Shamos's technique.

A presentation of the rotating calipers algorithm and a proof of its correctness with the same level of detail as the algorithms in the previous sections would require the development of a complex domain theory for geometric entities, and of the corresponding implementation primitives. Such a level of

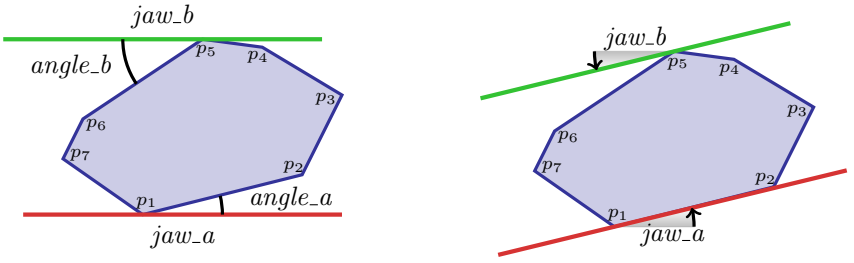
```
1  Levenshtein_distance (s, t: ARRAY [T]): INTEGER
2    require
3      s.lower = t.lower = 1
4      s.count = m
5      t.count = n
6    local
7      i, j: INTEGER
8      d: ARRAY [INTEGER, INTEGER]
9    do
10     d := {0}^(m+1) × {0}^(n+1)
11     across [1..m] as i loop d [i,0] := i end
12     across [1..n] as j loop d [0,j] := j end
13
14     across [1..m] as i
15     invariant
16       1 ≤ i ≤ m + 1
17       ∀h ∈ [0..i − 1], ∀k ∈ [0..m]: d [h, k] = distance (s [1..h], t [1..k])
18     loop
19       across [1..n] as j
20       invariant
21         1 ≤ i ≤ m + 1
22         1 ≤ j ≤ n + 1
23         ∀h ∈ [0..i − 1], ∀k ∈ [0..j − 1]: d [h, k] = distance (s [1..h], t [1..k])
24       loop
25         if s [i] = t [j] then
26           d [i,j] := d [i − 1, j − 1]
27         else
28           d [i,j] := 1 + min (d [i − 1, j − 1], d [i, j − 1], d [i − 1, j])
29         end
30       end
31     end
32     Result := d [m, n]
33   ensure
34     Result = distance (s, t)
35   end
```

Figure 18: Levenshtein distance.

detail is beyond the scope of this paper; instead, we outline the essential traits of the specification and give a description of the algorithm in pseudo-code in Figure 20.[6]

The algorithm inputs a list p of at least three points such that it represents a convex polygon (precondition on line 3) and returns the value of the polygon's

[6]The algorithm in Figure 20 is slightly simplified, as it does not deal explicitly with the special case where a line initially coincides with an edge: then, the minimum angle is zero, and hence the next vertex not on the line should be considered. This problem can be avoided by adjusting the initialization to avoid that a line coincides with an edge.



(a) Initial jaw configuration: antipodal pair (p_1, p_5).

(b) Jaws after one iteration: antipodal pair (p_2, p_5).

Figure 19: The rotating calipers algorithm illustrated.

diameter (postcondition on line 41).

It starts by adjusting the two parallel support lines on the two vertices with the maximum difference in y coordinate, such as p_1 and p_5 in Figure 19(a). Then, it enters a loop that computes the angle between each line and the next vertex on the polygon, and rotates both jaws by the minimum of such angles. At each iteration, it compares the distance between the new antipodal pair and the currently stored maximum (in **Result**), and updates the latter if necessary. In the example of Figure 19(a), *jaw_a* determines the smallest angle, and hence both jaws are rotated by *angle_a* in Figure 19(b). Such an informal description suggests the obvious bounding invariants that the two jaws are maintained parallel (line 18), **Result** varies between some initial value and the final value *diameter* (p) (line 19), and the total rotation of the calipers is between zero and $180+180$ (line 20). The essential invariant is, however, harder to state formally, because it involves a subset of the antipodal pairs reached by the calipers. A semi-formal presentation is:

$$\textbf{Result} = \max \left\{ |p_1, p_2| \;\middle|\; \begin{array}{l} p_1, p_2 \in p \quad \wedge \\ \mathit{reached}\ (p_1,\ \mathit{total\_rotation}) \quad \wedge \\ \mathit{reached}\ (p_2,\ \mathit{total\_rotation}) \end{array} \right\}$$

whose intended meaning is that **Result** stores the maximum distance between all points p_1, p_2 among p's vertices that can be reached with a rotation of up to *total_rotation* degrees from the initial calipers' horizontal positions.

4.6 Algorithms on data structures

Many data structures are designed around specific operations, which can be performed efficiently by virtue of the characterizing properties of the structures. This section presents linked lists and binary search trees and algorithms for some of such operations. The presentation of their invariants clarifies the connection between data-structure properties and the algorithms' correct design.

```
1  diameter_calipers (p: LIST [POINT]): INTEGER
2    require
3      p.count ≥ 3 ; p.is_convex
4    local
5      jaw_a, jaw_b: VECTOR -- Jaws of the caliper
6      n_a, n_b: NATURAL -- Pointers to vertices of the polygon
7      angle_a, angle_b: REAL -- Angle measures
8    do
9      n_a := "Index, in p, of the vertex with the minimum y coordinate"
10     n_b := "Index, in p, of the vertex with the maximum y coordinate"
11
12     jaw_a := "Horizontal direction from p[n_a] pointing towards negative"
13     jaw_b := "Horizontal direction from p[n_b] pointing towards positive"
14     from
15       total_rotation := 0
16       Result := |p[n_a] − p[n_b]| -- Distance between pair of vertices
17     invariant
18       parallel (jaw_a, jaw_b) -- Jaws are parallel
19       0 < Result ≤ diameter (p) -- Result increases until diameter(p)
20       0 ≤ total_rotation < 360
21     until total_rotation ≥ 180 -- All antipodal pairs considered
22     loop
23       angle_a := "Angle between p[n_a] and the next vertex in p "
24       angle_b := "Angle between p[n_b] and the next vertex in p "
25       if angle_a < angle_b then
26         -- Rotate jaw_a to coincide with the edge p[n_a]—p[n_a].next
27         jaw_a := jaw_a + angle_a
28         -- Rotate jaw_b by the same amount
29         jaw_b := jaw_b + angle_a
30         -- Next current point n_a
31         n_a := "Index of vertex following p[n_a]"
32         -- Update total rotation
33         total_rotation := total_rotation + angle_a
34       else
35         -- As in the then branch with a's and b's roles reversed
36       end
37       -- Update maximum distance between antipodal points
38       Result := max (|p[n_a] − p[n_b]|, Result)
39     variant 180 − total_rotation end
40   ensure
41     Result = diameter (p)
42   end
```

Figure 20: Diameter of a polygon with rotating calipers.

4.6.1 List reversal

Consider a list of elements of generic type G implemented as a linked list: each element's attribute *next* stores a reference to the next element in the list; and the last elements's *next* attribute is **Void**. This section discusses the classic algorithm that reverses a linked list iteratively.

We introduce a specification that abstracts some implementation details by means of a suitable domain theory. If *list* is a variable of type *LIST* [*G*]—that is, a reference to the first element—we lift the semantics of *list* in assertions to denote the *sequence* of elements found by following the chain of references until **Void**. This interpretation defines finite sequences only if *list*'s reference sequence is acyclic, which we write *acyclic* (*list*). Thus, the precondition of routine *reverse* is simply

$$\mathit{acyclic}\ (\ \mathit{list}\ ),$$

where *list* is the input linked list.

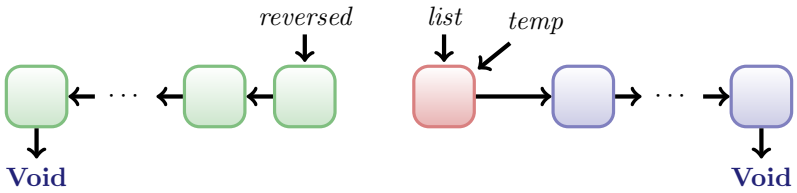
For a sequence $s = s_1\, s_2\, \ldots\, s_n$ of elements of length $n \geq 0$, its reversal *rev* (s) is inductively defined as:

$$\mathit{rev}(s) = \begin{cases} \epsilon & n = 0 \\ \mathit{rev}(s_2\, \ldots\, s_n) \circ s_1 & n > 0\,, \end{cases} \tag{46}$$

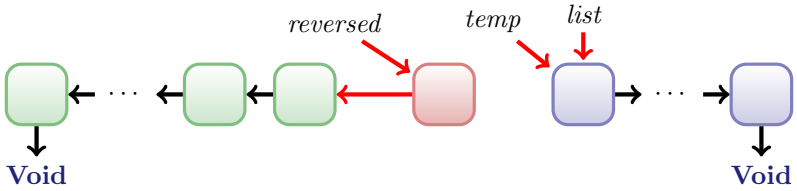
where ϵ denotes the empty sequence and "$\circ$" is the concatenation operator. With this notation, *reverse*'s postcondition is:

$$\mathit{list}\ = \mathit{rev}\ (\mathbf{old}\ \ \mathit{list}\ )$$

with the matching property that *list* is still acyclic.



(a) Before executing the loop body.



(b) After executing the loop body: new links are in red.

Figure 21: One iteration of the loop in routine *reverse*.

The domain theory for lists makes it possible to derive the loop invariant with the usual techniques. Let us introduce a local variable *reversed*, which will store the iteratively constructed reversed list. More precisely, every iteration of the loop removes one element from the beginning of *list* and moves it to the beginning of *reversed*, until all elements have been moved. When the loop terminates:

- *list* points to an empty list;
- *reversed* points to the reversal of **old** *list*.

Therefore, the routine concludes by assigning *reversed* to overwrite *list*. Backward substitution yields the loop's postcondition from the routine's:

$$reversed = rev\,(\textbf{old}\ \ list\,). \tag{47}$$

Using (46) for empty lists, we can equivalently write (47) as:

$$rev(\ list\,)\ \circ\ reversed = rev\,(\textbf{old}\ \ list\,), \tag{48}$$

which is the essential loop invariant, whereas $list = \textbf{Void}$ is the exit condition. The other component of the loop invariant is the constraint that *list* and *reversed* be acyclic, also by mutation of the postcondition.

Figure 22 shows the standard implementation. Figure 21 pictures instead a graphical representation of *reverse*'s behavior: Figure 21(a) shows the state of the lists in the middle of the computation, and Figure 21(b) shows how the state changes after one iteration, in a way that the invariant is preserved.

The correctness proof relies on some straightforward properties of the *rev* and $\circ$ functions. Initiation follows from the property that $s \circ \epsilon = s$. Consecution relies on the definition (46) for $n > 0$, so that:

$$rev\ (\ list\,.next)\ \circ\ \ list\,.\ first\ = rev\,(\ list\,).$$

Proving the preservation of acyclicity relies on the two properties:

$$\begin{array}{rcl} acyclic(s_1\, s_2\, \ldots\, s_n) & \Longrightarrow & acyclic(s_2\, \ldots\, s_n) \\ |s_1| = 1 \wedge\ acyclic(r) & \Longrightarrow & acyclic(s_1 \circ r). \end{array}$$

4.6.2 Binary search trees

Each node in a binary tree has at most two children, conventionally called *left* and *right*. Binary *search* trees are a special kind of binary trees whose nodes store values from some totally ordered domain T and are arranged reflecting the relative order of their values; namely, if t is a binary search tree and $n \in t$ is one of its nodes with value v, all nodes in n's left subtree store values less than or equal to v, and all nodes in n's right subtree store values greater than or equal to v. We express this characterizing property using domain-theory notation as:

$$\begin{array}{rcl} s \in t[\,n.\ left\,] & \Longrightarrow & s\,.\,value \leq n.value \\ s \in t[\,n.\,right\,] & \Longrightarrow & s\,.\,value \geq n.value\,, \end{array} \tag{49}$$

```
1  reverse ( list : LIST [G])
2    require
3      acyclic ( list )
4    local
5      reversed, temp: LIST [G]
6    do
7      from reversed := Void
8      invariant
9        rev ( list ) ∘ reversed = rev (old list )
10       acyclic ( list )
11       acyclic (reversed)
12     until list = Void
13     loop
14       temp := list .next
15       list .next := reversed
16       reversed := list
17       list := temp
18     variant list .count end
19     list := reversed
20   ensure
21     list = rev (old list )
22     acyclic ( list )
23   end
```

Figure 22: Reversal of a linked list.

where $t[n]$ denotes t's subtree rooted at node n. This property underpins the correctness of algorithms for operations such as searching, inserting, and removing nodes in binary search trees that run in time linear in a tree's *height*; for trees whose nodes are properly balanced, the height is logarithmic in the number of nodes, and hence the operations can be performed efficiently. We now illustrate two of these algorithms with their invariants.

Consider searching for a node with value *key* in a binary search tree t. If $t.\mathit{values}$ denotes the set of values stored in t, the specification of this operation consists of the postcondition

$$\mathit{key} \in t.\mathit{values} \implies \mathbf{Result} \in t \ \wedge \ \mathit{key} = \mathbf{Result}.\mathit{value} \tag{50}$$

$$\mathit{key} \notin t.\mathit{values} \implies \mathbf{Result} = \mathbf{Void}\,, \tag{51}$$

where **Void** is returned if no node has value *key*. For simplicity, we only consider non-empty trees—handling the special case of an empty tree is straightforward.

We can obtain the essential invariant by weakening both conjuncts in (50) based on two immediate properties of trees. First, $\mathbf{Result} \in t$ implies $\mathbf{Result} \neq \mathbf{Void}$, because no valid node is **Void**. Second, a node's value belongs to the set of values of the subtree rooted at the node:

$$n.\mathit{value} \ \in \ t[n].\mathit{values}$$

for $n \in t$. Thus, the following formula is a weakening of (50):

$$key \in t.values \implies \textbf{Result} \neq \textbf{Void} \wedge key \in t[\textbf{Result}].values, \quad (52)$$

which works as essential loop invariant for binary-search-tree search.

Search works by moving **Result** to the left or right subtree—according to (49)—until a value *key* is found or the subtree to be explored is empty. This corresponds to the disjunctive exit condition $\textbf{Result} = \textbf{Void} \vee \textbf{Result}.value = key$ and to the bounding invariant $\textbf{Result} \neq \textbf{Void} \implies \textbf{Result} \in t$: we are within the tree until we hit **Void**. Figure 23 shows the corresponding implementation.

```
1  has_bst (t: BS_TREE [T]; key: T): NODE
2    require
3      t.root ≠ Void -- nonempty tree
4    do
5      from Result := t.root
6      invariant
7        Result ≠ Void ⟹ Result ∈ t
8        key ∈ t.values ⟹ Result ≠ Void ∧ key ∈ t[Result].values
9      until Result = Void ∨ key = Result.value
10     loop
11       if key < Result.value then
12         Result := Result.left
13       else
14         Result := Result.right
15       end
16     end
17   ensure
18     key ∈ t.values ⟹ Result ∈ t ∧ key = Result.value
19     key ∉ t.values ⟹ Result = Void
20   end
```

Figure 23: Search in a binary search tree.

Initiation follows from the precondition and from the identity $t[t.root] = t$. Consecution relies on the following domain property, which in turn follows from (49):

$$n \in t \wedge v \in t[n].values \wedge v < n.value \implies n.left \neq \textbf{Void} \wedge v \in t[n.left].values$$
$$n \in t \wedge v \in t[n].values \wedge v > n.value \implies n.right \neq \textbf{Void} \wedge v \in t[n.right].values.$$

The ordering property (49) entails that the leftmost node in a (non-empty) tree t—that is the first node without left child reached by always going left from the root—stores the minimum of all node values. This property, expressible using the domain-theory function *leftmost* as:

$$\min(t.values) = leftmost(t).value, \quad (53)$$

leads to an algorithm to determine the node with minimum value in a binary search tree, whose postcondition is thus:

$$\textbf{Result} \;=\; \mathit{leftmost}(t) \tag{54}$$
$$\textbf{Result}.\mathit{value} \;=\; \min(t.\mathit{values})\,. \tag{55}$$

The algorithm only has to establish (54), which then implies (55) combined with the property (53). In fact, the algorithm is oblivious of (49) and operates solely based on structural properties of binary trees; (55) follows as an afterthought.

Duplicating the right-hand side of (54), writing t in the equivalent form $t[\,t.\mathit{root}\,]$, and applying constant relaxation to $t.\mathit{root}$ yields the essential invariant

$$\mathit{leftmost}(t[\textbf{Result}]) = \mathit{leftmost}(t) \tag{56}$$

with the bounding invariant $\textbf{Result} \in t$ that we remain inside the tree t. These invariants capture the procedure informally highlighted above: walk down the left children until you reach the leftmost node. The corresponding implementation is in Figure 24.

```
1  min_bst (t: BS_TREE [T]): NODE
2    require
3      t.root ≠ Void -- nonempty tree
4    do
5      from Result := t.root
6      invariant
7        Result ∈ t
8        leftmost (t[Result]) = leftmost (t)
9      until Result.left = Void
10     loop
11       Result := Result.left
12     end
13   ensure
14     Result = leftmost (t)
15     Result.value = min (t.values)
16   end
```

Figure 24: Minimum in a binary search tree.

Initiation follows by trivial identities. Consecution relies on a structural property of the leftmost node in any binary tree:

$$n \in t \wedge n.\,\mathit{left} \neq \textbf{Void} \quad\Longrightarrow\quad n.\,\mathit{left} \in t \wedge \mathit{leftmost}(t[n]) = \mathit{leftmost}(t[\,n.\,\mathit{left}\,]).$$

4.7 Fixpoint algorithms: PageRank

PageRank is a measure of the popularity of nodes in a network, used by the Google Internet search engine. The basic idea is that the PageRank score of a

node is higher the more nodes link to it (multiple links from the same page or self-links are ignored). More precisely, the PageRank score is the probability that a random visit on the graph (with uniform probability on the outgoing links) reaches it at some point in time. The score also takes into account a *dampening* factor, which limits the number of consecutive links followed in a random visit. If the graph is modeled by an adjacency matrix (modified to take into account multiple and self-links, and to make sure that there are no sink nodes without outgoing edges), the PageRank scores are the entries of the *dominant eigenvector* of the matrix.

In our presentation, the algorithm does not deal directly with the adjacency matrix but inputs information about the graph through arguments *reaching* and *outbound*. The former is an array of sets of nodes: *reaching* [k] denotes the set of nodes that directly link to node k. The other argument *outbound* [k] denotes instead the number of outbound links (to different nodes) originating in node k. The **Result** is a vector of n real numbers, encoded as an array, where n is the number of nodes. If *eigenvector* denotes the dominant eigenvector (also of length n) of the adjacency matrix (defined implicitly), the postcondition states that the algorithm computes the dominant eigenvector to within precision ϵ (another input):

$$|\mathit{eigenvector} - \mathbf{Result}| < \epsilon\,. \tag{57}$$

That is, **Result** [k] is the rank of node k to within overall precision ϵ.

The algorithm computes the PageRank score iteratively: it starts assuming a uniform probability on the n nodes, and then it updates it until convergence to a fixpoint. Before every iteration, the algorithm saves the previous values of **Result** as *old_rank*, so that it can evaluate the progress made after the iteration by comparing the sum *diff* of all pairwise absolute differences, one for each node, between the scores saved in *old_rank* and the newly computed scores available in **Result**. Correspondingly, the main loop's essential invariant is postcondition (57) with *diff* substituted for ϵ. *diff* gets smaller with every iteration of the main loop, until it becomes less than ϵ, the main loop terminates, and the postcondition holds. The connection between the main loop invariants and the postcondition is thus straightforward.

Figure 25 shows an implementation of this algorithm. The two across loops nested within the main loop update the PageRank scores in **Result**. Every iteration of the outer **across** loop updates the value of **Result** [i] for node i as:

$$\frac{(1 - \mathit{dampening})}{n} + \mathit{dampening} \cdot \sum_{j} \frac{\mathit{old\_rank}[j]}{\mathit{outbound}[j]}\,. \tag{58}$$

The inner loop computes the sum in (58) for j ranging over the set *reaching* [i] of nodes that directly reach i. The invariants of the across loops express the progress in the computation of (58); we do not write them down explicitly as they are not particularly insightful from the perspective of connecting postconditions and loop invariants.

```
page_rank (dampening, ϵ: REAL; reaching: ARRAY [SET [INTEGER]];
           outbound: ARRAY [INTEGER]): ARRAY [REAL]
  require
    0 < dampening <1
    ϵ >0
    reaching.count = outbound.count = n >0
  local
     diff : REAL
    old_rank: ARRAY [REAL]
     link_to : SET [INTEGER]
  do
    old_rank := {1/n}^n -- Initialized with n elements all equal to 1/n
    from diff := 1
    invariant
      | eigenvector − Result | < diff
    until diff <ϵ
    loop
       diff := 0
      across [1.. n] as i loop
        Result [i] := 0
         link_to := reaching [i]
        across [1.. link_to .count] as j loop
          Result [i] := Result [i] + old_rank [j] / outbound [j]
        end
        Result [i] := dampening ∗ Result [i] + (1 − dampening)/n
         diff := diff + |Result [i] − old_rank [i]|
      end
      old_rank := Result -- Copy values of Result into old_rank
    variant 1 + diff − ϵ
    end
  ensure
    | eigenvector − Result | < ϵ
  end
```

Figure 25: PageRank fixpoint algorithm.

5 Related work: Automatic invariant inference

The amount of research work on the automated inference of invariants is substantial and spread over more than three decades; this reflects the cardinal role that invariants play in the formal analysis and verification of programs. This section outlines a few fundamental approaches that emerged, without any pretense of being exhaustive. A natural classification of the methods to infer invariants is between *static* and *dynamic*. Static methods (Section 5.1) use only the program text, whereas dynamic methods (Section 5.2) summarize the properties of many program executions with different inputs.

Program construction In the introductory sections, we already mentioned classical formal methods for program construction [17, 26, 46, 48] on which this survey paper is based. In particular, the connection between a loop's postcondition and its invariant buttresses the classic methods of program construction; this survey paper has demonstrated it on a variety of examples. In previous work [23], we developed gin-pink, a tool that practically exploits the connection between postconditions and invariants. Given a program annotated with postconditions, gin-pink systematically generates mutations based on the heuristics of Section 3.2, and then uses the Boogie program verifier [43] to check which mutations are correct invariants. The gin-pink approach borrows ideas from both static and dynamic methods for invariant inference: it is only based on the program text (and specification) as the former, but it generates "candidate" invariants to be checked—like dynamic methods do.

Reasoning about domain theories To bridge the gap between the levels of abstraction of domain theories and of their underlying atomic assertions (see Section 2), one needs to reason about first- or even higher-logic formulas often involving interpreted theories such as arithmetic. The research in this area of automated theorem proving is huge; interested readers are referred to the many reference publications on the topic [56, 8, 6, 39].

5.1 Static methods

Historically, the earliest methods for invariant inference where *static* as in the pioneering work of Karr [35]. Abstract interpretation and the constraint-based approach are the two most widespread frameworks for static invariant inference (see also Bradley and Manna [**?**]Chap. 12]BM07-book). Jhala and Majumdar [34] provide an overview of the most important static techniques and discuss how they are applied in combination with different problems of program verification.

Abstract interpretation is a symbolic execution of programs over abstract domains that over-approximates the semantics of loop iteration. Since the seminal work by Cousot and Cousot [13], the technique has been updated and extended to deal with features of modern programming languages such as object-orientation and heap memory-management (e.g., Logozzo [44] and Chang and Leino [9]). One of the main successes of abstract interpretation has been the development of sound but incomplete tools [3] that can verify the absence of simple and common programming errors such as division by zero or void dereferencing.

Constraint-based techniques rely on sophisticated decision procedures over non-trivial mathematical domains (such as polynomials or convex polyhedra) to represent concisely the semantics of loops with respect to certain template properties.

Static methods are sound and often complete with respect to the class of invariants that they can infer. Soundness and completeness are achieved by leveraging the decidability of the underlying mathematical domains they represent; this implies that the extension of these techniques to new classes of

properties is often limited by undecidability. State-of-the-art static techniques can infer invariants in the form of mathematical domains such as linear inequalities [14, 11], polynomials [58, 57], restricted properties of arrays [7, 5, 29], and linear arithmetic with uninterpreted functions [2].

Following Section 3.1, the loop invariants that static techniques can easily infer are often a form of "bounding" invariant. This suggests that the specialized static techniques for loop invariant inference discussed in this section, and the idea of deriving the loop invariant from the postcondition, demonstrated in the rest of the paper, can be fruitfully combined: the former can easily provide bounding loop invariants, whereas the latter can suggest the "essential" components that directly connect to the postcondition.

To our knowledge, there are only a few approaches to static invariant inference that take advantage of existing annotations [52, 33, 16, 40, 38]. Janota [33] relies on user-provided assertions nested within loop bodies and tries to check whether they hold as invariants of the loop. The approach has been evaluated only on a limited number of straightforward examples. De Caso et al. [16] briefly discuss deriving the invariant of a "for" loop from its postcondition, within a framework for reasoning about programs written in a specialized programming language. Lahiri et al. [40] also leverage specifications to derive intermediate assertions, but focusing on lower-level and type-like properties of pointers. On the other hand, Păsăreanu and Visser [52] derive candidate invariants from postconditions within a framework for symbolic execution and model-checking.

Finally, Kovács and Voronkov [38] derive complex loop invariants by first encoding the loop semantics as recurring relations and then instructing a rewrite-based theorem prover to try to remove the dependency on the iterator variables in the relations. This approach exploits heuristics that, while do not guarantee completeness, are practically effective to derive automatically loop invariants with complex quantification—a scenario that is beyond the capabilities of most other methods.

5.2 Dynamic methods

Only in the last decade have dynamic techniques been applied to invariant inference. The Daikon approach of Ernst et al. [19] showed that dynamic inference is practical and sprung much derivative work (e.g., Perkings and Ernst [53], Csallner et al. [15], Polikarpova et al. [55], Ghezzi et al. [25], Wei et al. [63], Nguyen et al. [49], and many others). In a nutshell, the dynamic approach consists in testing a large number of candidate properties against several program runs; the properties that are not violated in any of the runs are retained as "likely" invariants. This implies that the inference is not sound but only an "educated guess": dynamic invariant inference is to static inference what testing is to program proofs. Nonetheless, just like testing is quite effective and useful in practice, dynamic invariant inference can work well if properly implemented. With the latest improvements [64], dynamic invariant inference can attain soundness of over 99% for the "guessed" invariants.

Among the numerous attempts to improve the effectiveness and flexibility of

dynamic inference, Gupta and Heidepriem [27] suggest to improve the quality of inferred contracts by using different test suites (based on code coverage and invariant coverage), and by retaining only the contracts that are inferred with both techniques. Fraser and Zeller [21] simplify and improve test cases based on mining recurring usage patterns in code bases; the simplified tests are easier to understand and focus on common usage. Other approaches to improve the quality of inferred contracts combine static and dynamic techniques [15, 61, 64].

To date, dynamic invariant inference has been mostly used to infer pre- and postconditions or intermediate assertions, whereas it has been only rarely applied [49] to *loop invariant* inference. This is probably because dynamic techniques require a sufficiently varied collection of test cases, and this is more difficult to achieve for loops—especially if they manipulate complex data structures.

6 Lessons from the mechanical proofs

Our verified Boogie implementations of the algorithms mirror the presentation in Section 4, as they introduce the predicates, functions, and properties of the domain theory that are directly used in the specification and in the correctness proofs. The technology of automatic theorem proving is, however, still in active development, and faces in any case insurmountable theoretical limits of undecidability. As a consequence, in a few cases we had to introduce explicitly some intermediate domain properties that were, in principle, logical consequences of other definitions, but that the prover could not derive without suggestion. Similarly, in other cases we had to guide the proof by writing down explicitly some of the intermediate steps in a form acceptable to the verifier.

A lesson of this effort is that the syntactic form in which definitions and properties of the domain theory are expressed may influence the amount of additional annotations required for proof automation. The case of the sorting algorithms in Section 4.3 is instructive. All rely on the definition of predicate *sorted* as:

$$\forall i \in [\, a.lower .. a.upper - 1]\colon\ a\ [\, i] \leq a\ [\, i\ + 1], \tag{59}$$

which compares *adjacent* elements in positions i and $i + 1$. Since bubble sort rearranges elements in an array also by swapping adjacent elements, proving it in Boogie was straightforward, since the prover could figure out how to apply definition (59) to discharge the various verification conditions—which also feature comparisons of adjacent elements. Selection sort and insertion sort required substantially more guidance in the form of additional domain theory properties and detailing of intermediate verification steps, since their logic does not operate directly on adjacent elements, and hence their verification conditions are syntactically dissimilar to (59). Changing the definition of *sorted* into something that relates non-adjacent elements—such as $\forall i, j : a.lower \leq i \leq j \leq a.upper \Longrightarrow a[i] \leq a[j]$—is not sufficient to bridge the semantic gap between sortedness and other predicates: the logic of selection sort and insertion sort remains more complex to reason about than that of bubble

sort. On the contrary, comb sort was as easy as bubble sort, because it relies on a generalization of *sorted* that directly matches its logic (see Section 4.3.6).

A similar experience occurred for the two dynamic programming algorithms of Section 4.4. While the implementation of Levenshtein distance closely mirrors the recursive definition of *distance* (Section 4.4.2) in computing the minimum, the relation between specification and implementation is less straightforward for the knapsack problem (Section 4.4.1), which correspondingly required a more complicated axiomatization for the proof in Boogie.

7 Conclusions and assessment

The concept of loop invariant is, as this review has attempted to show, one of the foundational ideas of software construction. We have seen many examples of the richness and diversity of practical loop invariants, and how they illuminate important algorithms from many different areas of computer science. We hope that these examples establish the claim, made at the start of the article, that the invariant is the key to every loop: to devise a new loop so that it is correct requires summoning the proper invariant; to understand an existing loop requires understanding its invariant.

Invariants belong to a number of categories, for which this discussion has established a classification which we hope readers will find widely applicable, and useful in understanding the loop invariants of algorithms in their own fields. The classification is surprisingly simple; perhaps other researchers will find new criteria that have eluded us.

Descriptions of algorithms in articles and textbooks has increasingly, in recent years, included loop invariants, at least informally; we hope that the present discussion will help reinforce this trend and increase the awareness—not only in the research community but also among software practitioners—of the centrality of loop invariants in programming.

Informal invariants are good, but being able to express them formally is even better. Reaching this goal and, more generally, continuing to make invariants ever more mainstream in software development requires convenient, clear and expressive notations for expressing loop invariants and other program assertions. One of the contributions of this article will be, we hope, to establish the practicality of domain-theory-based invariants, which express properties at a high level of abstraction, and their superiority over approaches that always revert to the lowest level (suggesting a possible slogan: "Quantifiers considered harmful").

Methodological injunctions are necessary, but the history of software practice shows that they only succeed when supported by effective tools. Many programmers still find it hard to come up with invariants, and this survey shows that they have some justifications: even though the basic idea is often clear, coming up with a sound and complete invariant is an arduous task. Progress in invariant inference, both theoretical and on the engineering side, remains essential. There is already, as noted, considerable work on this topic, often aimed at infer-

ring more general invariant properties than the inductive loop invariants of the present discussion; but much remains to be done to bring the tools to a level where they can be integrated in a standard development environment and routinely suggest invariants, conveniently and correctly, whenever a programmer writes a loop. The work mentioned in Section 5 is a step in this direction.

Another major lessons for us from preparing this survey (and reflecting on how different it is from what could have been written on the same topic 30, 20, 10 or even 5 years ago) came from our success in running essentially all the examples through formal, mechanized proofs. Verification tools such as Boogie, while still a work in progress, have now reached a level of quality and practicality that enables them to provide resounding guarantees for work that, in the past, would have remained subject to human errors.

We hope that others will continue this work, both on the conceptual side—by providing further insights into the concept of loop invariant—and on the practical side—by extending the concept to its counterpart for data (i.e., the class invariant) and by broadening our exploration and classification effort to many other important algorithms of computer science.

Acknowledgments This work is clearly based on the insights of Robert Floyd, C.A.R. Hoare and Edsger Dijkstra in introducing and developing the notion of loop invariant. The mechanized proofs were made possible by Rustan Leino's Boogie tool, a significant advance in turning axiomatic semantics into a practically usable technique. We are grateful to our former and present ETH colleagues Bernd Schoeller, Nadia Polikarpova, and Julian Tschannen for pioneering the use of Boogie in our environment.

References

[1] Samson Abramsky and Achim Jung. Domain theory. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science III*. Oxford University Press, 1994.

[2] Dirk Beyer, Thomas A. Henzinger, Rupak Majumdar, and Andrey Rybalchenko. Invariant synthesis for combined theories. In Byron Cook and Andreas Podelski, editors, *Proceedings of the 8th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'07)*, volume 4349 of *Lecture Notes in Computer Science*, pages 378–394. Springer, 2007.

[3] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A static analyzer for large safety-critical software. In *Proceedings of the 2003 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'03)*, pages 196–207. ACM, 2003.

[4] Joshua Bloch. Extra, extra – read all about it: Nearly all binary searches and mergesorts are broken. `http://googleresearch.blogspot.ch/2006/06/extra-extra-read-all-about-it-nearly.html`, 2006.

[5] Marius Bozga, Peter Habermehl, Radu Iosif, Filip Konečný, and Tomáš Vojnar. Automatic verification of integer array programs. In Ahmed Bouajjani and Oded Maler, editors, *Proceedings of the 21st International Conference on Computer Aided Verification (CAV'09)*, volume 5643 of *Lecture Notes in Computer Science*, pages 157–172. Springer, 2009.

[6] Aaron R. Bradley and Zohar Manna. *The Calculus of Computation*. Springer, 2007.

[7] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. What's decidable about arrays? In E. Allen Emerson and Kedar S. Namjoshi, editors, *Proceedings of the 7th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'06)*, volume 3855 of *Lecture Notes in Computer Science*, pages 427–442. Springer, 2006.

[8] Bruno Buchberger. Mathematical theory exploration. In *Automated Reasoning, Third International Joint Conference (IJCAR)*, volume 4130 of *Lecture Notes in Computer Science*, pages 1–2. Springer, 2006.

[9] Bor-Yuh Evan Chang and K. Rustan M. Leino. Abstract interpretation with alien expressions and heap structures. In Radhia Cousot, editor, *Proceedings of the 6th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'05)*, volume 3385 of *Lecture Notes in Computer Science*, pages 147–163. Springer, 2005.

[10] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 1999.

[11] Michael Colón, Sriram Sankaranarayanan, and Henny Sipma. Linear invariant generation using non-linear constraint solving. In Jr. Warren A. Hunt and Fabio Somenzi, editors, *Proceedings of the 15th International Conference on Computer Aided Verification(CAV'03)*, volume 2725 of *Lecture Notes in Computer Science*, pages 420–432. Springer, 2003.

[12] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 3rd edition, 2009.

[13] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th Annual ACM Symposium on Principles of Programming Languages (POPL'77)*, pages 238–252, 1977.

[14] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th Annual ACM Symposium on Principles of Programming Languages (POPL'78)*, pages 84–96, 1978.

[15] Christoph Csallner, Nikolai Tillman, and Yannis Smaragdakis. DySy: dynamic symbolic execution for invariant inference. In Wilhelm Schäfer, Matthew B. Dwyer, and Volker Gruhn, editors, *Proceedings of the 30th International Conference on Software Engineering (ICSE'08)*, pages 281–290. ACM, 2008.

[16] Guido de Caso, Diego Garbervetsky, and Daniel Gorín. Reducing the number of annotations in a verification-oriented imperative language. In *Proceedings of Automatic Program Verification*, 2009.

[17] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.

[18] Standard ECMA-367. Eiffel: Analysis, design and programming language, 2006.

[19] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions of Software Engineering*, 27(2):99–123, 2001.

[20] Robert W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science*, volume 19 of *Proceedings of Symposia in Applied Mathematics*, pages 19–32. American Mathematical Society, 1967.

[21] Gordon Fraser and Andreas Zeller. Exploiting common object usage in test case generation. In *IEEE Fourth International Conference on Software Testing, Verification and Validation (ICST'11)*, pages 80–89. IEEE Computer Society, 2011.

[22] Carlo A. Furia, Dino Mandrioli, Angelo Morzenti, and Matteo Rossi. *Modeling Time in Computing*. Monographs in Theoretical Computer Science. An EATCS series. Springer, 2012.

[23] Carlo A. Furia and Bertrand Meyer. Inferring loop invariants using postconditions. In Andreas Blass, Nachum Dershowitz, and Wolfgang Reisig, editors, *Fields of Logic and Computation: Essays Dedicated to Yuri Gurevich on the Occasion of His 70th Birthday*, volume 6300 of *Lecture Notes in Computer Science*, pages 277–300. Springer, August 2010.

[24] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, 1979.

[25] Carlo Ghezzi, Andrea Mocci, and Mattia Monga. Synthesizing intensional behavior models by graph transformation. In *Proceedings of the 31st International Conference on Software Engineering (ICSE'09)*, pages 430–440. IEEE, 2009.

[26] David Gries. *The science of programming*. Springer-Verlag, 1981.

[27] Neelam Gupta and Zachary V. Heidepriem. A new structural coverage criterion for dynamic detection of program invariants. In *18th IEEE International Conference on Automated Software Engineering (ASE'03)*, pages 49–59. IEEE Computer Society, 2003.

[28] John Hatcliff, Gary T. Leavens, K. Rustan M. Leino, Peter Müller, and Matthew J. Parkinson. Behavioral interface specification languages. *ACM Comput. Surv.*, 44(3):16, 2012.

[29] Thomas A. Henzinger, Thibaud Hottelier, Laura Kovács, and Andrei Voronkov. Invariant and type inference for matrices. In *Proceedings of the 11th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'10)*, Lecture Notes in Computer Science. Springer, 2010.

[30] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[31] C. A. R. Hoare. Proof of correctness of data representations. *Acta Inf.*, 1:271–281, 1972.

[32] C. A. R. Hoare. The verifying compiler: A grand challenge for computing research. *Journal of the ACM*, 50(1):63–69, 2003.

[33] Mikoláš Janota. Assertion-based loop invariant generation. In *Proceedings of the 1st International Workshop on Invariant Generation (WING'07)*, 2007.

[34] Ranjit Jhala and Rupak Majumdar. Software model checking. *ACM Computing Surveys*, 41(4), 2009.

[35] Michael Karr. Affine relationships among variables of a program. *Acta Informatica*, 6:133–151, 1976.

[36] Hans Kellerer, Ulrich Pferschy, and David Pisinger. *Knapsack problems*. Springer, 2004.

[37] Donald E. Knuth. *The Art of Computer Programming (volumes 1–4A)*. Addison-Wesley, 2011. First edition: 1968–1973.

[38] Laura Kovács and Andrei Voronkov. Finding loop invariants for programs over arrays using a theorem prover. In Marsha Chechik and Martin Wirsing, editors, *Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering (FASE'09)*, volume 5503 of *Lecture Notes in Computer Science*, pages 470–485. Springer, 2009.

[39] Daniel Kroening and Ofer Strichman. *Decision Procedures: An Algorithmic Point of View*. Monographs in Theoretical Computer Science. An EATCS series. Springer, 2008.

[40] Shuvendu K. Lahiri, Shaz Qadeer, Juan P. Galeotti, Jan W. Voung, and Thomas Wies. Intra-module inference. In Ahmed Bouajjani and Oded Maler, editors, *Proceedings of the 21st International Conference on Computer Aided Verification (CAV'09)*, volume 5643 of *Lecture Notes in Computer Science*, pages 493–508. Springer, 2009.

[41] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, 1977.

[42] Leslie Lamport. Teaching concurrency. *SIGACT News*, 40(1):58–62, 2009.

[43] K. Rustan M. Leino. This is Boogie 2. (Manuscript KRML 178) `http://research.microsoft.com/en-us/projects/boogie/`, June 2008.

[44] Francesco Logozzo. Automatic inference of class invariants. In Bernhard Steffen and Giorgio Levi, editors, *Proceedings of the 5th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'04)*, volume 2937 of *Lecture Notes in Computer Science*, pages 211–222. Springer, 2004.

[45] Kurt Mehlhorn and Peter Sanders. *Algorithms and Data Structures: The Basic Toolbox*. Springer, 2008.

[46] Bertrand Meyer. A basis for the constructive approach to programming. In Simon H. Lavington, editor, *Proceedings of IFIP Congress 1980*, pages 293–298, 1980.

[47] Bertrand Meyer. *Object-oriented software construction*. Prentice Hall, 2nd edition, 1997. First Ed.: 1988.

[48] Carroll Morgan. *Programming from Specifications*. Prentice Hall, 2nd edition, 1994.

[49] ThanhVu Nguyen, Deepak Kapur, Westley Weimer, and Stephanie Forrest. Using dynamic analysis to discover polynomial and array invariants. In *34th International Conference on Software Engineering (ICSE'12)*, pages 683–693. IEEE, 2012.

[50] Susan Owicki and David Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6:319–340, 1976.

[51] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1993.

[52] Corina S. Păsăreanu and Willem Visser. Verification of Java programs using symbolic execution and invariant generation. In *Proceedings of the 11th International SPIN Workshop on Model Checking Software*, volume 2989 of *Lecture Notes in Computer Science*, pages 164–181. Springer, 2004.

[53] Jeff H. Perkings and Michael D. Ernst. Efficient incremental algorithms for dynamic detection of likely invariants. In Richard N. Taylor and Matthew B. Dwyer, editors, *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT'04/FSE-12)*, pages 23–32. ACM, 2004.

[54] André Platzer. Differential-algebraic dynamic logic for differential-algebraic programs. *J. Log. Comput.*, 20(1):309–352, 2010.

[55] Nadia Polikarpova, Ilinca Ciupa, and Bertrand Meyer. A comparative study of programmer-written and automatically inferred contracts. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'09)*, pages 93–104, 2009.

[56] Alan Robinson and Andrei Voronkov, editors. *Handbook of Automated Reasoning*. Elsevier, 2001.

[57] Enric Rodríguez-Carbonell and Deepak Kapur. Generating all polynomial invariants in simple loops. *Journal of Symbolic Computation*, 42(4):443–476, 2007.

[58] Sriram Sankaranarayanan, Henny Sipma, and Zohar Manna. Non-linear loop invariant generation using Gröbner bases. In Neil D. Jones and Xavier Leroy, editors, *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'04)*, pages 318–329. ACM, 2004.

[59] Michael I. Shamos. *Computational geometry*. PhD thesis, Yale University, 1978. `http://goo.gl/XiXNl`.

[60] J. Michael Spivey. *Z Notation – a reference manual*. Prentice Hall International Series in Computer Science. Prentice Hall, 2nd edition, 1992.

[61] Nikolai Tillmann, Feng Chen, and Wolfram Schulte. Discovering likely method specifications. In *8th International Conference on Formal Engineering Methods (ICFEM'06)*, volume 4260 of *Lecture Notes in Computer Science*, pages 717–736, 2006.

[62] Godfried T. Toussaint. Solving geometric problems with the rotating calipers. In *Proceedings of IEEE MELECON 1983, Athens, Greece*, 1983. `http://goo.gl/nnnc4`.

[63] Yi Wei, Carlo A. Furia, Nikolay Kazmin, and Bertrand Meyer. Inferring better contracts. In Richard N. Taylor, Harald Gall, and Nenad Medvidović, editors, *Proceedings of the 33rd International Conference on Software Engineering (ICSE'11)*, pages 191–200. ACM, May 2011.

[64] Yi Wei, Hannes Roth, Carlo A. Furia, Yu Pei, Alexander Horton, Michael Steindorfer, Martin Nordio, and Bertrand Meyer. Stateful testing: Finding

more errors in code and contracts. In Perry Alexander, Corina Pasareanu, and John Hosking, editors, *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE'11)*, pages 440–443. ACM, November 2011.