How to avoid proving the absence of integer overflows

Martin Clochard, Jean-Christophe Filliâtre, Andrei Paskevich

How to avoid proving the absence of integer overflows*

Martin Clochard[1,2], Jean-Christophe Filliâtre[1,2], and Andrei Paskevich[1,2]

[1] Lab. de Recherche en Informatique, Univ. Paris-Sud, CNRS, Orsay, F-91405
[2] INRIA Saclay – Île-de-France, Orsay, F-91893

Abstract. When proving safety of programs, we must show, in particular, the absence of integer overflows. Unfortunately, there are lots of situations where performing such a proof is extremely difficult, because the appropriate restrictions on function arguments are invasive and may be hard to infer. Yet, in certain cases, we can relax the desired property and only require the absence of overflow during the first n steps of execution, n being large enough for all practical purposes. It turns out that this relaxed property can be easily ensured for large classes of algorithms, so that only a minimal amount of proof is needed, if at all. The idea is to restrict the set of allowed arithmetic operations on the integer values in question, imposing a "speed limit" on their growth. For example, if we repeatedly increment a 64-bit integer, starting from zero, then we will need at least 2^{64} steps to reach an overflow; on current hardware, this takes several hundred years. When we do not expect any single execution of our program to run that long, we have effectively proved its safety against overflows of all variables with controlled growth speed. In this paper, we give a formal explanation of this approach, prove its soundness, and show how it is implemented in the context of deductive verification.

1 Introduction

Proving the safety of a program involves showing the absence of arithmetic overflows. By itself, an overflow does not crash the program, but it silently results in a meaningless value with, typically, fatal consequences in other places of the program. A famous example is described in Joshua Bloch's blog post *Nearly All Binary Searches and Mergesorts are Broken* [1]. It is related to the computation of the mean of two 32-bit array indices involved in these two algorithms in the Java standard library. When using large arrays, an arithmetic overflow may occur, possibly resulting in a negative array index and hence a program crash.

Today, most formal methods do tackle arithmetic overflows, *e.g.*, abstract interpretation [2], model checking [3], or deductive verification [4]. In deductive verification, for instance, one models machine integers as specific data types where operations are given suitable statically-verified preconditions to prevent overflows. In the case of binary search, it can be proved that the computation of the mid-point of `low` and `high` as `low + (high - low)/2` does not overflow.

* Work partly supported by the Joint Laboratory ProofInUse (ANR-13-LAB3-0007, `http://www.spark-2014.org/proofinuse`) of the French national research organization.

Yet there are many situations where it is extremely difficult to prove the absence of arithmetic overflows. Perhaps the simplest example is that of a global counter that is incremented by one every time a fresh value is requested, for instance to generate labels or timestamps. Deductive verification, in principle, could accommodate the necessary bound pre-conditions; yet such bounds would invade specifications throughout the program, resulting in an impractical annotation/proof burden.

We may, however, observe, that to overflow this global counter—assuming it is stored as a 64-bit unsigned integer and starts from zero—we need to perform 2^{64} individual increment operations, plus all the work we do on each new value. Even if we perform one billion increment operations per second, it would take us more than 584 years to reach the limit. Unless we expect our program to have century-long runs, we can rest assured that this particular counter is, for all intents and purposes, safe from overflow. The crucial part of the argument is that the counter can only grow by one at a time: arbitrary additions, multiplications and so on are not allowed. In this paper, we propose a way to make this meta-argument formal, and we prove the soundness of our approach. We also demonstrate an implementation of this method in Why3 [5], a tool for deductive program verification.

We stress that this approach does not reduce the need in traditional methods (such as deductive verification, abstract interpretation, or model checking) for proving the absence of integer overflows, as they target different uses of integers. The idea is to use our technique in combination with other methods, within the same program. We give an example of such a combination in this paper.

The paper is organized as follows. Section 2 motivates our work with classes of programs where we want to avoid exhibiting bounds on integers to prove the absence of overflows. Section 3 introduces our solution, along with its proof of soundness. Section 4 describes our implementation in Why3 and illustrates it with a representative example. We conclude with a discussion.

2 Motivating Examples

We have already mentioned the example of a symbol generator (*gensym* for short): a program returning a fresh integer on each call. A gensym is trivially implemented with a global variable, as described in Program 1. (A more robust implementation would hide the global variable s, using for instance a static variable local to function GENSYM.)

Program 1 Symbol generator

```
1: s ← 0
2: function GENSYM
3:     s ← s + 1
4:     return s
5: end function
```

Suppose that we want to prove, in a usual way, that the increment operation in GENSYM does not overflow the counter. In the context of deductive verification, we have

to put a bound pre-condition on GENSYM, requiring s to be strictly less than $2^{64} - 1$. In order to satisfy this precondition, we now have to constrain, in the same way, every user of GENSYM in our program. In cases where we call GENSYM inside a loop or a recursive function, new preconditions and invariants are needed in order to put a bound on the number of iterations or recursive calls. Essentially, we have to unroll the whole execution of our program and come up with sufficient bounds on its input in order to satisfy the bound pre-condition of GENSYM. At the very least, these added annotations will inflate the program specification and hamper verification. What is worse, inferring suitable bounds for the program input might be a computationally hard problem. However, if s is a 64-bit unsigned integer, and if we agree, as explained above, to content ourselves with the absence of overflows *during the first hundred years* of the program execution—suddenly, we have nothing to prove about GENSYM itself, and we only have to ensure that counter s is not modified in some dangerous way (say, doubled) elsewhere in the program.

A large class of examples, where one might want to apply this meta-argument, is that of programs computing the size of a data structure. Consider, for instance, a function computing the length of a linked list. It can be implemented either recursively (Program 2) or iteratively using a while loop (Program 3). In both cases, it amounts to incrementing the length by one for each list element. Just as in the previous case, if the result is stored in a 64-bit integer, it would take too much time to overflow it.

Program 2 Length of a list, recursive

```
1: function LENGTH(l)
2:     if l = null then return 0
3:     else return 1 + LENGTH(l.next)
4:     end if
5: end function
```

Program 3 Length of a list, iterative

```
1: function LENGTH(l)
2:     len ← 0
3:     while l ≠ null do
4:         len ← len + 1
5:         l ← l.next
6:     end while
7:     return len
8: end function
```

Another example in this category is Program 4 which computes the size of a binary tree, recursively. In this code, we compute the *sum* of the sizes of the two sub-trees, as returned by the recursive calls. Yet, in terms of computation, this addition is equivalent to a sequence of increments by one. This equivalence is evident if we rewrite the program to accumulate the size in a global counter.

Program 4 Size of a tree

```
1: function SIZE(t)
2:     if t = empty then return 0
3:     else return 1 + SIZE(t.left) + SIZE(t.right)
4:     end if
5: end function
```

Data structures with sharing are a special case. Consider for instance the tree of depth h shown on the right. It has size $2^h - 1$. If, when computing its size, we employ memoization in order to exploit the sharing and speed up computation, our meta-argument does not hold anymore. In that case, the additions performed could overflow since results of previous computations are reused and thus accumulated several times in the result.

However, if memoization is not used, our approach applies even for trees with sharing. Indeed, if we call the SIZE function from Program 4 on a tree with sharing, it will simply not terminate in practice for a large value of h, say 100, even if we were able to build that tree in space and time $O(h)$. This shows that our meta-argument is really about *time* (in this case, the time spent in the traversal of the data structure) and not *space* (the space used to store the data structure).

Another class of programs for which our approach applies is that of data structures that store integers for internal management. An example is non-empty linked lists with destructive concatenation, as illustrated in Fig. 1. Each list contains its length and two pointers to its first and last elements. When performing the concatenation of lists a and b, the last element of a now points to the first element of b. The pointer to the last element of a is updated so that it now points to the last element of b. Finally, the length of list a is updated and list b is invalidated, by setting its pointers to *null*. The code is given in Program 5.

Just as in the previous program, the addition on line 3 is not dangerous. Indeed, the length of each list is limited by the time spent to build it. Since list b is invalidated when performing APPEND(a,b), we are moving the "time credits" earned by the construction of b into the updated length of a, without risking an overflow.

Program 5 Destructive Append on Linked Lists

```
1: procedure APPEND(a, b)
2:     assert a ≠ b ∧ a.first ≠ null ∧ b.first ≠ null
3:     a.size ← a.size + b.size
4:     a.last.next ← b.first
5:     a.last ← b.last
6:     b.first ← null
7:     b.last ← null
8: end procedure
```

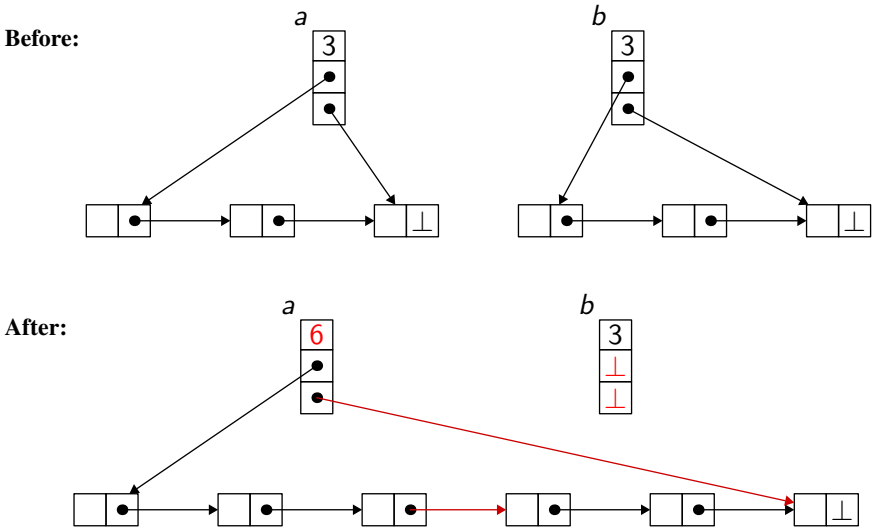

Fig. 1. Destructive append on linked lists.

Another example involving integers stored in a data structure is union-find with weighted union. A union-find structure implements equivalence classes with canonical representatives. It is a forest, where each node contains a pointer *link* to its father and an integer w, its weight, which is an upper bound of the length of a path from this node to a leaf. When performing the union of two classes represented by root nodes a and b, the weights are used to decide whether a is linked to b or, conversely, b is linked to a. When the weights are equal, we choose arbitrarily and we increment the weight of the root node by one. There is obviously no danger of arithmetic overflow here, since weights are only obtained by successive increments by one.

Program 6 Union-Find with Weighted Union

```
1: procedure UNION(a, b)
2:     if a ≠ b then
3:         if a.w < b.w then a.link ← b else b.link ← a end if
4:         if a.w = b.w then a.w ← a.w + 1 end if
5:     end if
6: end procedure
```

Another potential use case, which we do not consider in detail here, is that of de Bruijn indices [6], a technical solution to the implementation of binders in symbolic computation. A variable is represented as an integer, which counts the number of binders under which it appears. When performing substitutions, such integer must be updated, being either incremented or decremented. Such increments only occur one by one, hence are safe.

Program 7 Number of solutions to the N-queens problem

```
 1: function N-QUEENS(b)
 2:     if board b contains N queens then return 1
 3:     else
 4:         r ← 0
 5:         for any legal board b′ obtained by adding a queen to board b do
 6:             r ← r + N-QUEENS(b′)
 7:         end for
 8:         return r
 9:     end if
10: end function
```

Our final class of examples are combinatorics programs in which backtracking is used to enumerate all the solutions of a problem. Consider for instance the famous n-queens problem, where we count the number of ways to place n non-attacking queens on a $n \times n$ board. Program 7 contains pseudo-code for a function N-QUEENS that progressively adds queens to a board b until a solution is found. The numbers manipulated in this program are bounded by the number of solutions that have already been enumerated by the recursive calls.

What is important in this example is that we do not know any reasonable upper bounds on the value that we compute, so it would not be feasible to prove the absence of arithmetic overflow. Another, more useful, example in this category is the computation of Littlewood-Richardson coefficients [7].

3 A Solution

We propose the following method for verifying the safety against arithmetic overflows in programs like those enumerated in the previous section.

First of all, we identify, inside the program code, the integer values for which we do not want to (or realistically can not) exhibit bounds. For any such value, represented as an n-bit machine integer, we relax the safety property and seek to prove that overflow is impossible during the first 2^n execution steps (with respect to a reasonable operational semantics, where execution steps translate to a proportional number of processor cycles). We leave to the user the responsibility of verifying that the relaxed property is sufficient, that is, execution duration of this magnitude indeed exceeds all practical expectations. We believe that 64-bits integers fit perfectly our use case. On one hand, they are natively supported by modern general-purpose CPUs; on the other hand, we are not aware of any application where a single execution of a sequential program is expected to run long enough to be able to overflow a 64-bit integer with singular increment operations.

To ensure the relaxed safety property for the selected integer values, we restrict the operations available for manipulating these integers so as to prohibit arbitrary growth. Depending on the programming language and the verification methods, this restriction can be achieved in various ways. For example, we can introduce new data types for such

"restricted integers" and let the type system verify that only the permitted operations are used on them. Of course, these integers are still compiled to native machine integers. This is the approach we show in this paper. Alternatively, one can work with a program where the native integer types are used indiscriminately, and use a static analysis procedure to detect the values for which the restrictions are respected. Notice that this static analysis is itself, in essence, some form of type inference.

Below, we describe two kinds of such restricted integers. The first are *Peano integers*, where only the increment operation is available to produce a number greater than those already reached. Using Peano integers, we can implement programs 1 (symbol generator), 2 and 3 (length of a list), and 6 (union-find). The second class allows us to use addition as well, provided that we do not let the same number to be used more than once. To this end, we introduce *one-time integers*, where operations like addition invalidate their arguments. Using one-time integers, we can implement the remaining examples, namely programs 4 (tree size), 5 (destructive append), and 7 (n-queens).

3.1 Peano integers

Peano integers are introduced as a new data type *peano*. We provide a constant *zero* and a successor operation on that type:

$$\begin{array}{l} \mathit{zero} : \ \mathit{peano} \\ \mathit{succ}(p : \mathit{peano}) : \mathit{peano} \end{array}$$

Other operations on Peano integers are provided, such as the conversion into an arbitrary-precision integer

$$\mathit{to\_int}(p : \mathit{peano}) : \mathbb{Z}$$

or the construction of a Peano integer from an arbitrary integer x, provided it is no greater than a Peano integer p we have already built:

$$\mathit{cap}(x : \mathbb{Z},\ p : \mathit{peano}) : \mathit{peano}$$

This is not an exhaustive list. For instance, it is safe to compare two Peano integers, to compute the minimum or the maximum of two of them, etc.

3.2 One-time integers

One-time integers are introduced as a new data type *onetime*. To account for the idea that one-time integers are used in a *linear way* or, equivalently, in a destructive way, each one-time integer carries a Boolean validity flag in addition to its value. The validity flag is mutable, and is changed from *true* to *false* when the one-time integer is invalidated. The value itself is immutable, and the allowed operations generate fresh one-time integers.

For one-time integers, we provide zero and successor functions:

$$\begin{array}{l} \mathit{fresh\_zero}() : \ \mathit{onetime} \\ \mathit{succ}(p : \mathit{onetime}) : \mathit{onetime} \end{array}$$

Contrary to Peano integers, *fresh_zero* is not a constant, since it must return a new one-time integer distinct from all others. Function *succ* is also different from the Peano version: it requires its argument to be valid, destroys its validity, and returns a new, valid one-time integer. One-time integers also feature a destructive addition:

$$add(x : onetime,\ y : onetime) : onetime$$

As function *succ*, function *add* requires valid arguments and destroys them. Additionally, it requires x and y to be *distinct* one-time integers, to prevent one from doubling the value of a one-time integer. Finally, one-time integers can be non-destructively turned into Peano integers:

$$to\_peano(x : onetime) : peano$$

Notice that we provide no operation to check the validity state of a one-time integer in a program; the validity flag is used for verification purposes only. Since the compiler ought to translate one-time integers to native machine integers, the validity of a given one-time integer cannot be available at the run time. For the same reason, we provide no operation to check whether two one-time integers are physically the same object.

3.3 Formalization

To prove the soundness of our approach, we consider a small While-like programming language with heap-allocated records. Figure 2 introduces the abstract syntax of this language, which distinguishes values v, expressions e, and statements s. Values are either Peano integers, memory addresses, Booleans, or arbitrary-precision integers. The set of addresses is assumed to be infinite. A one-time integer is represented as a record with two fields: a field *otP* containing its value, as a Peano integer; and a field *otV* containing its validity flag. As a consequence, operation *to_peano*, which turns a one-time integer into a Peano integer, is simply a field access. Access to the *otV* field, though, is forbidden, for the reasons explained above.

We equip our language with a small-step operational semantics. A heap Σ is a finite-domain partial mapping from address/field pairs to values. A program state is a triple (V,Σ,s), where V is a mapping from variables to values, Σ is a heap, and s is a statement representing the remaining execution. The operational semantics defines a one-step execution for expressions, written $V,\Sigma,e \to V,\Sigma',e'$, as well as a one-step execution for statements, $V,\Sigma,s \to V',\Sigma',s'$. As usual, such relations are defined with head reductions (Fig. 3–5) and reduction contexts (Fig. 6). Note that expressions do not modify the variable store V. Standard reduction rules for arbitrary-precision integers and Booleans are omitted, for the sake of brevity. It is worth pointing out that direct construction and mutation of one-time integers is not allowed; see rules ALLOC and MEM-ASSIGN.

Informally, the main theorem can be stated as follows: a program that contains no non-zero Peano constants will not cause any Peano/one-time integer to exceed n in its first n steps of execution. Formally, we first define a notion of bounded program states:

$v ::= \langle p \rangle$		Peano integer ($p \in \mathbb{N}$)
	$\mid a$	memory address
	$\mid \bot \mid \top$	Boolean value
	$\mid n$	arbitrary-precision integer ($n \in \mathbb{Z}$)
$e ::= x$		variable
	$\mid v$	value
	$\mid succ(e)$	Peano/one-time successor
	$\mid fresh\_zero()$	fresh one-time integer
	$\mid add(e,e)$	one-time addition
	$\mid to\_int(e)$	conversion from Peano to arbitrary integers
	$\mid cap(e,e)$	conversion from integers to Peano (partial)
	$\mid \{f = e, \ldots, f = e\}$	record construction
	$\mid e.f$	field access
	$\mid \ldots$	Boolean connectives, operations on arbitrary integers, etc.
$s ::= skip$		skip
	$\mid x \leftarrow e$	variable assignment
	$\mid e.f \leftarrow e$	memory assignment
	$\mid s; s$	sequence
	$\mid$ `if` e `then` s `else` s	conditional
	$\mid$ `while` e `do` s `done`	loop

Fig. 2. Abstract syntax for a small programming language.

Definition 1. *A state (V,Σ,s) is n-bounded if:*

- *Any Peano integer occurring anywhere in the state, including constants in the program s, is no greater than n;*
- *The sum of all valid one-time integers allocated in Σ is no greater than n.*

Then we can state the main result. It uses the notion of 0-bounded state to capture the idea that all Peano integers are zeros at the start of a program.

Theorem 1. *Let (V,Σ,s) be a 0-bounded state. Then, for any state (V',Σ',s') reachable after n steps of execution, (V',Σ',s') is n-bounded.*

Proof. First, we generalize the claim: if (V,Σ,s) is m-bounded (with $m \geq 0$), then for any state (V',Σ',s') reachable after n steps of execution, (V',Σ',s') is $(m+n)$-bounded. By a straightforward induction on the number of steps, we reduce to the case of a single step of execution. Then we proceed by case analysis on the head reduction rule:

- PEANO-SUCC: as the Peano integer is bounded by m, its successor is bounded by $m+1$. Other parts of the state do not change, so the resulting state is indeed $(m+1)$-bounded.
- ONE-TIME-SUCC: Similar to the rule PEANO-SUCC, except that the sum of valid one-time integers also changes. However, it increases by exactly one, so the resulting state is indeed $(m+1)$-bounded.

$$\text{VAR}\ \frac{}{V, \Sigma, x \ \rightarrow\ V, \Sigma, V(x)} \qquad \text{FIELD}\ \frac{a.f \in \Sigma \qquad f \neq otV}{V, \Sigma, a.f \ \rightarrow\ V, \Sigma, \Sigma(a.f)}$$

$$\text{ALLOC}\ \frac{a \notin \Sigma \qquad \forall i.\, f_i \notin \{otP, otV\}}{V, \Sigma, \{f_1 = v_1, \ldots, f_n = v_n\} \ \rightarrow\ V, \Sigma[a.f_1 \leftarrow v_1, \ldots, a.f_n \leftarrow v_n], a}$$

Fig. 3. Reduction rules for expressions.

$$\text{SKIP}\ \frac{}{V, \Sigma, (skip\,; s) \ \rightarrow\ V, \Sigma, s} \qquad \text{ASSIGN}\ \frac{}{V, \Sigma, x \leftarrow v \ \rightarrow\ V[x \leftarrow v], \Sigma, skip}$$

$$\text{MEM-ASSIGN}\ \frac{a.f \in \Sigma \qquad f \notin \{otP, otV\}}{V, \Sigma, a.f \leftarrow v \ \rightarrow\ V, \Sigma[a.f \leftarrow v], skip}$$

$$\text{IF-TRUE}\ \frac{}{V, \Sigma, (\texttt{if}\ \top\ \texttt{then}\ s\ \texttt{else}\ s') \ \rightarrow\ V, \Sigma, s}$$

$$\text{IF-FALSE}\ \frac{}{V, \Sigma, (\texttt{if}\ \bot\ \texttt{then}\ s\ \texttt{else}\ s') \ \rightarrow\ V, \Sigma, s'}$$

$$\text{WHILE}\ \frac{}{V, \Sigma, (\texttt{while}\ e\ \texttt{do}\ s\ \texttt{done}) \ \rightarrow\ V, \Sigma, (\texttt{if}\ e\ \texttt{then}\ s\,;\ \texttt{while}\ e\ \texttt{do}\ s\ \texttt{done}\ \texttt{else}\ skip)}$$

Fig. 4. Reduction rules for statements.

$$\text{PEANO-SUCC}\ \frac{}{V, \Sigma, succ(\langle p \rangle) \ \rightarrow\ V, \Sigma, \langle p+1 \rangle}$$

$$\text{ONE-TIME-SUCC}\ \frac{\Sigma(a) = \{otP = \langle p \rangle, otV = \top\} \qquad a' \notin \Sigma}{V, \Sigma, succ(a) \ \rightarrow\ V, \Sigma[a.otV \leftarrow \bot, a'.otP \leftarrow \langle p+1 \rangle, a'.otV \leftarrow \top], a'}$$

$$\text{FRESH\_ZERO}\ \frac{a \notin \Sigma}{V, \Sigma, fresh\_zero() \ \rightarrow\ V, \Sigma[a.otP \leftarrow \langle 0 \rangle, a.otV \leftarrow \top], a}$$

$$\text{TO\_INT}\ \frac{}{V, \Sigma, to\_int(\langle n \rangle) \ \rightarrow\ V, \Sigma, n} \qquad \text{CAP}\ \frac{0 \leq n \leq m}{V, \Sigma, cap(n, \langle m \rangle) \ \rightarrow\ V, \Sigma, \langle n \rangle}$$

$$\text{ADD}\ \frac{\Sigma(a_1) = \{otP = \langle n \rangle, otV = \top\} \quad \Sigma(a_2) = \{otP = \langle m \rangle, otV = \top\} \quad a_1 \neq a_2 \quad a_3 \notin \Sigma}{V, \Sigma, add(a_1, a_2) \ \rightarrow\ V, \Sigma[a_1.otV \leftarrow \bot, a_2.otV \leftarrow \bot, a_3.otP \leftarrow \langle n+m \rangle, a_3.otV \leftarrow \top], a_3}$$

Fig. 5. Reduction rules for operations.

$$
\begin{array}{l}
C_e ::= \Box \mid succ(C_e) \\
\quad \mid add(C_e, e) \mid add(e, C_e) \\
\quad \mid int(C_e) \\
\quad \mid cap(C_e, e) \mid cap(e, C_e) \\
\quad \mid \{f = e, \ldots, f = C_e, \ldots, f = e\} \\
\quad \mid C_e.f \\
\quad \mid \ldots \qquad \text{(other usual reduction contexts, for Boolean connectives/etc.)} \\
\\
C_s ::= \Box \mid x \leftarrow C_e \\
\quad \mid C_e.f \leftarrow e \mid e.f \leftarrow C_e \\
\quad \mid C_s; s \\
\quad \mid \texttt{if } C_e \texttt{ then } s \texttt{ else } s \\
\quad \mid \texttt{while } C_e \texttt{ do } s \texttt{ done}
\end{array}
$$

Fig. 6. Reduction contexts.

- FRESH_ZERO: the newly introduced one-time integer is 0, so the total sum of valid one-time integers stays unchanged. As it is itself trivially m-bounded, the resulting state is m-bounded as well, hence $(m+1)$-bounded.
- CAP: by hypothesis, it creates a Peano integer smaller than an existing one, thus respecting the bound.
- ADD: the total sum of valid integers stays unchanged and is therefore m-bounded. All we need to show is that the result of the addition is no greater than $m+1$. Using the separation hypothesis, the result is no greater than the total sum of valid one-time integers in the initial state, hence no greater than m.
- MEM-ASSIGN: it does not introduce any new Peano nor one-time integer. Also, one-time integers are not modified by this rule, so the sum of valid one-time integers is still no greater than m.
- ALLOC: since this rule cannot be used to build a one-time integer, the resulting state is still m-bounded.
- other rules: As they do not introduce new Peano/one-time integers and do not change the memory, they preserve m-boundedness. □

Note that in the proof above, the bound may only increase after successor operations. This yields the following corollary:

Corollary 1. *For any execution $(V,\Sigma,s) \rightarrow^\star (V',\Sigma',s')$, where (V,Σ,s) is a 0-bounded state, (V',Σ',s') is bounded by the number of successor steps in the execution (rules* PEANO-SUCC *and* ONE-TIME-SUCC*).*

4 Implementation in Why3

Why3 is a platform for deductive program verification. It provides a rich language, called WhyML, to write programs [5] and their logical specifications [8, 9], and it relies on external theorem provers to discharge verification conditions. Why3 is based on first-order logic with rank-1 polymorphic types, algebraic data types, inductive predicates,

and several other extensions. The programming language can be seen as an ML dialect, providing variant types, pattern matching, exceptions, and mutable data structures. In order to keep proof obligations reasonably easy to read and to debug, Why3 imposes static control of aliases: every l-value in a program must have a finite set of names and these names must be known at the time of generation of verification conditions.

Verified WhyML programs can be automatically translated to OCaml, producing executable correct-by-construction code. This procedure, called *code extraction*, is guided by *drivers*: configuration files which assign OCaml translation to symbols that have not been given definition in the WhyML program. During extraction, Why3 erases from the program so-called *ghost code* which serves to facilitate specification and verification and is guaranteed to not affect the observable program behaviour and its final result [10]. For example, a ghost function argument can be used to pass a witness of some existential pre-condition; a ghost record field may hold a pure logical "view" of the record's contents.

Here is how Peano and one-time integers are introduced in Why3.

```
type peano model { v: int }

type onetime model { peano: peano; mutable valid: bool }
```

Here, type `int` is that of mathematical, arbitrary-precision integers. Both types are introduced as *model types* whose structure is hidden from programs but can be accessed from specification annotations. By virtue of being model types, their values cannot be constructed in programs directly: the client code has to employ abstract functions. For instance, addition over Peano integers can be implemented via the basic abstract operation `cap` as follows:

```
val cap (x: int) (p: peano) : peano requires { 0 <= x <= p.v }
                                    ensures  { result.v = x }

let add (p q r: peano) : peano requires { 0 <= p.v + q.v <= r.v }
                              ensures  { result.v = p.v + q.v }
  = cap (to_int p + to_int q) r
```

Notice that addition takes a third argument, serving as an upper bound for the result. For comparison, addition over one-time integers is specified as follows:

```
val add (o1 o2: onetime) : onetime writes { o1, o2 }
  requires { o1.valid ∧ o2.valid }
  ensures  { result.peano.v = o1.peano.v + o2.peano.v }
  ensures  { result.valid ∧ not o1.valid ∧ not o2.valid }
```

Notice that we do not need to add any separation pre-condition for the arguments of `add`: Why3 assumes it by default and checks separation whenever `add` is called.

To produce executable OCaml code, Why3 provides appropriate driver files that translate Peano and one-time integers directly to OCaml's native unboxed 63-bit integers on 64-bit architectures. For instance, Peano's operation `add` is translated into the OCaml function `(fun p q _ → p + q)`. Notice that the translation does not contain

any run-time safety assertion: the bound argument is simply ignored. Deductive verification of the initial WhyML program ensures, statically, that the precondition of `add` is satisfied in any possible execution.

Similarly, operation `add` for one-time integers is translated into OCaml native addition `(+)`. It is worth pointing out that the validity flag appears nowhere in the extracted code. Indeed, once the safety of calls to operations over one-time integers has been established during the verification phase, the validity bit has no further influence on the program behaviour and can be eliminated.

Example. To illustrate the use of our approach, let us consider implementing Program 7 in Why3. Figure 7 shows a code where backtracking is implemented with recursive function `count_bt_queens`. Its argument `solutions` is a Peano counter, which is incremented each time we find a new solution (line 22). For the sake of brevity, we omit specification annotations.

This program uses two flavors of machine integers. Type `int63` is used for array indexes and chessboard coordinates and we prove statically the safety of operations on this type. For example, at line 12, a proof obligation is generated to ensure that `q` can be incremented without overflow. This obligation is easily discharged, thanks to the loop condition. The other flavor is Peano integers, which we use to count the solutions; see the incrementation at line 22. No proof obligation is generated for this operation. This is fortunate, since we would not be able to prove it. Indeed, we do not have any *a priori* bound on the number of solutions, except the obviously too large $n!$-related ones. And any bound inferred automatically (*e.g.*, by abstract interpretation) would be even larger. Both types `int63` and `peano` become OCaml type `int` in the extracted 64-bit code.

Caveats. It may seem that the bounds passed to functions `cap` and `add` for Peano integers should be ghost arguments, as they only serve the verification purposes and are ignored in the extracted OCaml code. Indeed, it is tempting to specify `add` as

```
val bad_add (p q: peano) (ghost r: peano) : peano
  requires { 0 <= p.v + q.v <= r.v }
  ensures { result.v = p.v + q.v }
```

and simplify the translation to `"(+)"`. This, however, would compromise the safety of the Peano integers, because a client code could write a ghost loop, incrementing some ghost variable up to an arbitrarily big value, well beyond 2^{64}. This ghost loop incurs zero run-time expense, as it is erased during extraction. Yet the ghost variable can be used in a call to `bad_add`, giving a "false alibi" to an overflowing non-ghost integer. In future versions of Why3, we may work around this problem by forbidding calls to `succ` in ghost code, making `bad_add` safe to use.

It should also be noted that we must not provide a function converting a `peano` value to a fixed-size machine integer. Just as in the previous case, as long as `succ` is admitted in ghost code, we can create an out-of-bounds ghost Peano value. Converting it to a 64-bit integer would lead to a contradiction (*i.e.*, a proof of `false`) in a reachable state of execution, compromising all subsequent verification conditions. Moreover, such conversion is dangerous even in the case where the offending value is not ghost. Indeed, while we may consider the states after 2^{64} steps as effectively unreachable (which

```
1   exception Inconsistent
2
3   let check_is_consistent (board: array int63) (pos: int63)
4   = try
5       let q = ref (of_int 0) in
6       while !q < pos do
7         let bq   = board[!q]   in
8         let bpos = board[pos] in
9         if bq        = bpos     then raise Inconsistent;
10        if bq - bpos = pos - !q then raise Inconsistent;
11        if bpos - bq = pos - !q then raise Inconsistent;
12        q := !q + of_int 1
13      done;
14      True
15    with Inconsistent →
16      False
17    end
18
19  let rec count_bt_queens (solutions: ref peano)
20      (board: array int63) (n: int63) (pos: int63)
21  = if eq pos n then
22      solutions := Peano.succ !solutions
23    else
24      let i = ref (of_int 0) in
25      while !i < n do
26        board[pos] ← !i;
27        if check_is_consistent board pos then
28          count_bt_queens solutions board n (pos + of_int 1);
29        i := !i + of_int 1
30      done
31
32  let count_queens (board: array int63) (n: int63) : peano
33  = let solutions = ref (Peano.zero ()) in
34    count_bt_queens solutions board n (of_int 0);
35    !solutions
```

Fig. 7. N-queens in Why3.

justifies the contradiction), it is disturbing if the system validates the total functional correctness, termination included, of

```
let p = ackermann_with_peano 4 2 (* = 2^65536 - 3 *) in
let n = to_int64 p in
assert { n > max_int64 >= n }
```

without raising any red flags.

On the other hand, it is perfectly safe to provide variants of the `cap` function taking fixed-size integers as first argument. For programs that work with native integers, this avoids an unnecessary conversion to arbitrary-precision integers.

5 Conclusion

We have presented a method to avoid proving the absence of arithmetic overflows when we do not expect a single execution to run long enough to overflow a machine integer of a fixed width. To the best of our knowledge, this is the first practical approach to verifying safety of programs such as the ones listed in Sec. 2. Despite its sheer simplicity, we feel that it effectively addresses a real-life verification challenge.

Our technique consists in placing an upper bound on reachable integer values in a sequential program, as a function of the number of execution steps. To apply this technique safely, a number of conditions must be taken into account. The most obvious one is the ratio of the chosen integer size to the available processor speed. We believe that 64-bit integers are a good match for the modern hardware.

If the program is written in a compiled language, one also needs to be aware of compiler optimisations. If a compiler rewrites **for** $i = 1$ **to** 2^{32} **do** $s \leftarrow s + 1$ **end for** into $s \leftarrow s + 2^{32}$ then our meta-argument clearly does not hold anymore. In practice, this should not be a concern for reasonably written algorithms. We observe that most of the time, if not always, the use of function `succ` on Peano/one-time integers coincides with an allocation or a branching point in the program, and thus is not amenable to an aggressive optimization. To reduce doubt, one can instrument the `succ` operation with some kind of "barrier instruction" that the compiler is not allowed to optimize out.

Computation on multiple cores introduces some additional constraints. For example, it would be unsound to add one-time integers computed on different cores: the individual increments are not serialized in this case, and the addition should produce either an arbitrary-precision integer or a fixed-width integer with a run-time check.

The solution proposed in this paper is not readily applicable when we want to use short machine integers (of 8 bits or less). For example, when implementing balanced binary search trees with AVL [11], the height of the sub-tree is stored inside each node. The height of an AVL tree of n nodes does not exceed $1.44 \log_2(n)$. Thus 6 bits for the height would allow up to 2^{44} nodes and 8 bits for the height would allow up to 2^{177} nodes. The same argument applies to the union-find weights. One possible approach to this problem is to introduce a variation of one-time integers where the linear quantity (a subject to increments and destructive additions; for example, the size of a tree) is stored as a ghost field, and the desired logarithmic quantity (tree height) is stored as a non-ghost field linked to the linear quantity by a statically verified datatype invariant. Devising a suitable generic interface for this purpose is one future direction of this work.

It is tempting to apply our approach for physical limits other than time, such as available memory or energy. This is however not straightforward. It is not enough to impose a physical limit; we must also be able to verify, statically, that a particular integer value in the program grows "in lockstep" with the consumption of the resource. So far, we were not able to ensure such a property for any non-trivial use case.

Acknowledgments. We are grateful to Arthur Charguéraud for detailed and constructive comments regarding a first draft of this paper.

References

1. Bloch, J.: Nearly all binary searches and mergesorts are broken (2006) `http://googleresearch.blogspot.com/2006/06/extra-extra-read-all-about-it-nearly.html`.
2. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: The Astrée static analyzer `http://www.astree.ens.fr/`.
3. Cordeiro, L., Fischer, B., Marques-Silva, J.: SMT-based bounded model checking for embedded ANSI-C software. In: Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering. ASE '09, Washington, DC, USA, IEEE Computer Society (2009) 137–148
4. Tuch, H., Klein, G., Norrish, M.: Types, bytes, and separation logic. In Hofmann, M., Felleisen, M., eds.: Proc. 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'07), Nice, France (January 2007) 97–108
5. Filliâtre, J.C., Paskevich, A.: Why3 — where programs meet provers. In Felleisen, M., Gardner, P., eds.: Proceedings of the 22nd European Symposium on Programming. Volume 7792 of Lecture Notes in Computer Science., Springer (March 2013) 125–128
6. de Bruijn, N.G.: Lambda calculus with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. Proc. of the Koninklijke Nederlands Akademie **75**(5) (1972) 380–392
7. Littlewood, D., Richardson, A.: Group Characters and Algebra. Philosophical transactions of the Royal Society of London: Mathematical and physical sciences. Harrison & Sons (1934)
8. Bobot, F., Filliâtre, J.C., Marché, C., Paskevich, A.: Why3: Shepherd your herd of provers. In: Boogie 2011: First International Workshop on Intermediate Verification Languages, Wrocław, Poland (August 2011) 53–64
9. Filliâtre, J.C.: One logic to use them all. In: 24th International Conference on Automated Deduction (CADE-24). Volume 7898 of Lecture Notes in Artificial Intelligence., Lake Placid, USA, Springer (June 2013) 1–20
10. Filliâtre, J.C., Gondelman, L., Paskevich, A.: The spirit of ghost code. In Biere, A., Bloem, R., eds.: 26th International Conference on Computer Aided Verification. Volume 8859 of Lecture Notes in Computer Science., Vienna, Austria, Springer (July 2014) 1–16
11. Adel'son-Vel'skiĭ, G.M., Landis, E.M.: An algorithm for the organization of information. Soviet Mathematics–Doklady **3**(5) (September 1962) 1259–1263