Chapter 6

Separation Logic

Separation logic was introduced around year 2000 by Reynolds and O'Hearn [6]. There are indeed several variants of this logic [1, 8]. It is implemented in several tools like Smallfoot [2] and Verifast [7], that use specific, partially automated provers as back-ends, or Ynot [5] or CFML [3] that uses Coq to perform the proofs.

6.1 Basics of Separation Logic

6.1.1 Syntax of Programs

We do not consider any encoding of pointers and memory heap like the Component-as-Array model of Section 5.2.3, so we directly consider the language introduced in Section 5.2.1. But, to make the formalization of memory access uniform, we remove mutable variables from that language: every mutable data must now be of the form of a pointer to a record, and only record fields are mutable. This is indeed how the references in OCaml or in Why3 are introduced: these are not built-in in the language but defined using a record with one mutable field. We detail this in Section 6.2.

For declaring such records, we keep the same syntax as in previous chapter.

$$\textsf{record}\ id = \{f_1 : \tau_1; \cdots f_n : \tau_n\}$$

We also generalize this language in order to deal with memory allocation and deallocation. The important point is how to extend the language of formulas in specifications, which is detailed in Section 6.1.3.

The syntax of expressions is now as follows.

$$\begin{array}{rcll} e & ::= & ... & \text{former expression constructs} \\ & & & \text{(without mutable variables)} \\ & & \mid e \rightarrow f & \text{field access} \\ & & \mid e \rightarrow f := e & \text{field update} \\ & & \mid \texttt{new}\ S & \text{allocation} \\ & & \mid \texttt{dispose}\ e & \text{deallocation} \end{array}$$

The expression `new` S allocates a new record of type S, and returns a pointer to it. The expression `dispose` e deallocates the pointer denoted by expression e, and returns $()$.

6.1.2 Operational Semantics

To support allocation and deallocation, we modify the operational semantics given in Section 5.2.2. Instead of considering the memory heap as a *total* map from pairs (loc,field name) to values, we consider that a heap is any *partial* map from (loc,field name) to values. We need to introduce a few notations: if h, h_1, h_2 are such partial maps:

- $\mathsf{dom}(h)$ denotes the *domain* of h, i.e. the set of pairs (loc,field name) where it is defined ;
- we write $h = h_1 \oplus h_2$ to mean that h is the disjoint union of h_1 and h_2, i.e.
  - $\mathsf{dom}(h_1) \cap \mathsf{dom}(h_2) = \emptyset$
  - $\mathsf{dom}(h) = \mathsf{dom}(h_1) \cup \mathsf{dom}(h_2)$
  - $h(l, f) = h_1(l, f)$ if $(l, f) \in \mathsf{dom}(h_1)$
  - $h(l, f) = h_2(l, f)$ if $(l, f) \in \mathsf{dom}(h_2)$

Note that in the following, the domain of partial heaps are always finite.

The operational semantics is defined by the relation

$$h, \Pi, e \rightsquigarrow h', \Pi', e'$$

where h is a partial heap. The rules are as follows.

Field access

$$\frac{l, f \in \mathsf{dom}(h) \qquad h(l, f) = v}{h, \Pi, (l \rightarrow f) \rightsquigarrow h, \Pi, v}$$

Implicitly, this rule means that if $(l, f) \notin \mathsf{dom}(h)$ then the execution cannot be done, it "blocks". Typically in practice this corresponds to an invalid memory access, which usually stops the program execution with a message like "segmentation fault" or "memory fault"

Field update

$$\frac{(l, f) \in \mathsf{dom}(h) \qquad h' = h\{(l, f) \leftarrow v\}}{h, \Pi, (l \rightarrow f := v) \rightsquigarrow h', \Pi, ()}$$

Again, if $(l, f) \notin \mathsf{dom}(h)$ then the execution blocks.

Allocation

$$\frac{l \notin \mathsf{dom}(h) \qquad h' = h \oplus \{(l, f) \leftarrow \mathit{default}(\tau) \mid f : \tau \in S\}}{h, \Pi, (\texttt{new}\ S) \rightsquigarrow h', \Pi, l}$$

The premise $l \notin \mathsf{dom}(h)$ means that l is "fresh", i.e. it is not yet allocated in h. The other premise expresses the initialization of the allocated fields depending on their types, i.e. $\mathit{default}(\mathit{int}) = 0$, $\mathit{default}(\mathit{bool}) = \mathit{false}$, $\mathit{default}(\mathit{real}) = 0.0$, $\mathit{default}(S) = \mathit{null}$.

Notice that from this definition, the allocation never blocks, our formalization does not consider memory overflow issues.

Deallocation

$$\frac{\text{for all } f \text{ of } S,\ (l, f) \in \mathsf{dom}(h) \qquad h' = h \backslash l}{h, \Pi, (\texttt{dispose}\ l) \rightsquigarrow h', \Pi, ()}$$

where the notation $h' = h \backslash l$ means $h'(l', f') = h(l', f')$ if $l' \neq l$, undefined otherwise. Notice that deallocation blocks if one attempts to deallocate a memory location that is not allocated.

Example 6.1.1 *The small program below illustrates our operational semantics.*

```
record List = { data : int, next: List }
```

$[], []$
`let x = new List in` $\rightsquigarrow$
$[(l_0, data) = 0, (l_0, next) = null], [x = l_0]$
`x→next := new List;` $\rightsquigarrow$
$[(l_0, data) = 0, (l_0, next) = l_1, (l_1, data) = 0, (l_1, next) = null], [x = l_0]$
`dispose(x→next);` $\rightsquigarrow$
$[(l_0, data) = 0, (l_0, next) = l_1], [x = l_0]$
`x→next→data := 1`
execution blocks

6.1.3 Separation Logic Formulas

A important idea in Separation Logic is that the language of terms remains *unchanged*, in particular the expression $e \to f$ *is not a term.* The language of formulas is extended with

- special atoms specifying *available memory ressources*
- a special connective called *separating* conjunction

The new grammar for formulas is as follows.

$$\begin{array}{rcll} P, Q & ::= & ... & \text{former constructs} \\ & & \mid \mathsf{emp} & \text{empty heap} \\ & & \mid t_1 \overset{f}{\mapsto} t_2 & \text{memory chunk} \\ & & \mid P * Q & \text{separating conjunction} \end{array}$$

where

- t_1 is a term of type S for some record type S
- f is a field of type τ in S
- t_2 is a term of type τ

These three new constructs allow to describe *finite portions of the memory heap*

The semantics of formulas is defined as usual by an interpretation $[\![P]\!]_{h,\Pi}$ that now depends on a partial heap h.

- Special formula emp:

$$[\![\mathsf{emp}]\!]_{h,\Pi} \text{ valid iff } \mathsf{dom}(h) = \emptyset$$

- Memory chunk: $[\![t_1 \overset{f}{\mapsto} t_2]\!]_{h,\Pi}$ iff
  - $[\![t_1]\!]_\Pi = l$ for some location l
  - $\mathsf{dom}(h) = \{(l, f)\}$
  - $h(l, f) = [\![t_2]\!]_\Pi$
- Separating conjunction: $[\![P * Q]\!]_{h,\Pi}$ is valid iff
  there exists h_1, h_2 such that

– $h = h_1 \oplus h_2$

– $[\![P]\!]_{h_1,\Pi}$ is valid

– $[\![Q]\!]_{h_2,\Pi}$ is valid

The semantics of the separating conjunction explicitly requires that the two conjuncts are valid on disjoint heaps. This means that predicates like our former predicates `disjoint` and `no_repet` are in some sense *internalized* in the logic.

Example 6.1.2 *Assuming two non-null locations l_0 and l_1, the stack $\Pi = [x = l_0]$ and the following partial heaps*

$$\begin{aligned} h_1 &= [(l_0, next) = l_1] \\ h_2 &= [(l_0, next) = l_1, (l_0, data) = 42] \\ h_3 &= [(l_0, next) = l_1, (l_1, next) = null] \end{aligned}$$

the table below gives the validity of a few formulas (Y for valid, N otherwise).

valid in ?	h_1	h_2	h_3
$x \overset{next}{\mapsto} l_1$	Y	N	N
$x \overset{next}{\mapsto} l_1 * x \overset{data}{\mapsto} 42$	N	Y	N
$x \overset{next}{\mapsto} l_1 * l_1 \overset{next}{\mapsto} null$	N	N	Y
emp	N	N	N

Properties of Separating Conjunction Separating conjunction satisfies the following identities

- $(P * Q) * R \leftrightarrow P * (Q * R)$
- $P * Q \leftrightarrow Q * P$
- $\mathsf{emp} * P \leftrightarrow P$
- if P, Q are memory-free formulas, $P * Q \leftrightarrow P \wedge Q$

In general P not equivalent to $P * P$. In fact this identity is true only if P is emp or a memory-free formula. This is a *linearity* aspect of the separating conjunction.

Symbolic Heaps The semantics given above has some issues regarding memory-free formulas. For example, using the heaps of Example 6.1.2 above, even if the formula $x \overset{next}{\mapsto} l_1$ is valid only in heap h_1, the conjunction $x \overset{next}{\mapsto} l_1 * true$ is valid also in heap h_1 and h_2. A variant in the semantics would be to consider that a memory-free formula is valid only in an empty heap.

Going further, a classical fragment of separation logic is the so-called *Symbolic Heaps*. Only formulas of the form

$$\exists v_1, \ldots, v_n, P_1 * P_2 * \cdots * P_k * \phi$$

are considered, where ϕ is a memory-free formula and the P_i are memory chunks.

In the rest of this chapter, we consider this fragment only, with the semantics that memory-free formulas are valid only in an empty heap.

Example 6.1.3 *The following function resets the field `f` of some structure to zero.*

```
record S { f : int }

function reset_f(x:S):unit
  requires  $\exists v. x \overset{f}{\mapsto} v$
  ensures  $x \overset{f}{\mapsto} 0$
  body x→f := 0
```

Since one cannot directly use `x→f` *as a term, an extra quantification is needed to specify that* `x→f` *is allocated.*

The following function increments the field `f` *by one.*

```
function incr_f(x:S) (ghost v:int):unit
  requires  $x \overset{f}{\mapsto} v$
  ensures  $x \overset{f}{\mapsto} v + 1$
  body x→f := x→f + 1
```

A ghost variable is needed to talk about the old value of `x→f`*.*

Some syntactic sugar are usually used: existentially quantified variables starting with an underscore, and quantification made implicit.

Example 6.1.4 *The same functions as in previous example, with syntactic sugar*

```
function reset_f(x:S):unit
  requires  $x \overset{f}{\mapsto} \_$
  ensures  $x \overset{f}{\mapsto} 0$
  body x→f := 0

function incr_f(x:S) :unit
  requires  $x \overset{f}{\mapsto} \_v$
  ensures  $x \overset{f}{\mapsto} \_v + 1$
  body x→f := x→f + 1
```

Notice how the ghost variable is made implicit.

6.1.4 Separation Logic Reasoning Rules

We can define rules for reasoning on programs in the similar style as Hoare rules, on triples $\{P\}e\{Q\}$ where P and Q are symbolic heaps.

The following rules can easily be proved correct.

Field access

$$\frac{}{\{l \overset{f}{\mapsto} v\} l \to f \{l \overset{f}{\mapsto} v * \mathsf{result} = v\}}$$

Field update

$$\frac{}{\{l \overset{f}{\mapsto} v\} l \to f := v' \{l \overset{f}{\mapsto} v' * \mathsf{result} = ()\}}$$

Allocation

$$\frac{}{\{\mathsf{emp}\}\texttt{new}\ S\{\_l \overset{f_1}{\mapsto} \mathit{default}(\tau_1) * \cdots * \_l \overset{f_n}{\mapsto} \mathit{default}(\tau_n) * \mathsf{result} = \_l\}}$$

where S is declared as $\{f_1 : \tau_1; \cdots f_n : \tau_n\}$

Deallocation

$$\frac{}{\{l \overset{f_1}{\mapsto} v_1 * \cdots * l \overset{f_n}{\mapsto} v_n\}\texttt{dispose}\ l\{\mathsf{emp}\}}$$

where l has type S.

Frame rule A very important rule that can be stated is the frame rule, which allows to reason *locally*:

$$\frac{\{P\}e\{Q\}}{\{P * R\}e\{Q * R\}}$$

in particular it states that whenever $\{P\}e\{Q\}$ is proven valid, its effects are confined in the partial heaps that are described by P and Q, and consequently it remains valid in any context R which is disjoint from P and Q.

Separation Logic and Symbolic Execution It has been noticed that reasoning with Separation Logic rules is a kind of *symbolic execution* [1]. Thanks to the frame rule, a proof of a program made of a sequence $e_1; \cdots ; e_n$ can be presented under the form

$$\{P_0\}e_1\{P_1\}e_2 \cdots e_n\{P_n\}$$

where at each step i we have $P_{i-1} = Q * R$, $P_i = Q' * R$ and $\{Q\}e_i\{Q'\}$ is valid for a rule above.

Example 6.1.5 *Here is a simple example of Separation Logic proof in the form of symbolic execution.*

$\{\mathit{emp}\}$
let *x = new List* ***in***
$\{(\_l \overset{data}{\mapsto} 0) * (\_l \overset{next}{\mapsto} \mathit{null}) * (x = \_l)\}$
$\{(x \overset{data}{\mapsto} 0) * (x \overset{next}{\mapsto} \mathit{null})\}$ *(consequence)*
x→data := 42 ;
$\{(x \overset{data}{\mapsto} 42) * (x \overset{next}{\mapsto} \mathit{null})\}$ *(frame)*
x→next := new List ;
$\{(x \overset{data}{\mapsto} 42) * (x \overset{next}{\mapsto} \_l) * (\_l \overset{data}{\mapsto} 0) * (\_l \overset{next}{\mapsto} \mathit{null})\}$

6.2 Case of References à la OCaml

The mutable variables that we had in previous chapters but we dropped in our new language, can be simulated using a pointer to a record with one field only. For convenience it is made polymorphic.

```
record Ref α = { contents : α }

function ref (x: α) : Ref α
  body let r = new Ref in r→contents := x; r
```

```
function (!) (r: Ref $\alpha$) : $\alpha$
  body r$\rightarrow$contents

function (:=) (r: Ref $\alpha$) (x:$\alpha$) : unit
  body r$\rightarrow$contents := x
```

These functions can be given natural specifications in separation logic. For readability we abbreviate $r \overset{contents}{\mapsto} t$ into $r \mapsto t$

```
function ref (x: $\alpha$) : Ref $\alpha$
  requires  emp
  ensures  result $\mapsto x$

function (!) (r: Ref $\alpha$) : $\alpha$
  requires  $r \mapsto \_v$
  ensures  $r \mapsto \_v *$ result $= \_v$

function (:=) (r: Ref $\alpha$) (x:$\alpha$) : unit
  requires  $r \mapsto \_$
  ensures  $r \mapsto x$
```

6.3 Case of Linked Lists

Similarly as the case of Component-as-Array model, we can define linked data structures using inductive predicates. For linked list, we can define

inductive $ls(List, List)$ =
 | ls_nil: $\forall x : List, ls(x, x)$
 | ls_cons: $\forall xyz : List, (x \overset{next}{\mapsto} y) * ls(y, z) \rightarrow ls(x, z)$

The important novelty is the use of a separating conjunction in the second case, which makes explicit that the tail of the list is disjoint from its head, and thus it has no repetition.

During a Separation Logic proof, one should apply purely logic reasoning steps using general lemmas like

$$ls(x, y) \leftrightarrow x = y \vee \exists z, (x \overset{next}{\mapsto} z) * ls(z, y)$$

From such lemmas one can derives the so-called *symbolic execution rules*, e.g.

$$ls(x, y) * x \neq y \rightsquigarrow (x \overset{next}{\mapsto} \_z) * ls(\_z, y) * x \neq y$$

In practice, systems implementing separation logic use a specific reasoning engine that applies such rules (they do not use off-the-shelf SMT provers).

Example: in-place list reversal

We consider again the example of in-place linked list reversal. We simplicity, we do not specify that the result list is the reverse of the input, but only that the output is a null-terminated list if the input is such a list. This avoids the need for ghost variables. The specified program we want to prove is thus the following (we use our "references" defined in previous section).

```
function reverse (l:List) : List =
  requires ls(l,null)
  ensures ls(result,null)
  body
   let p = ref l in
   let r = ref null in
   while !p ≠ null do
     invariant p→_lp ∗ ls(_lp,null) ∗ r→_lr ∗ ls(_lr,null)
     let n = !p→next in
     !p→next := r;
     r := !p;
     p := n;
  done;
  !r
```

Notice that the loop invariant tells that lists $!p$ and $!r$ are null-terminated and disjoint.

The first part of the proof is to show by symbolic execution that the loop invariant is initially true. This is as follows.

$\{ls(l, null)\}$
`let p = ref l in`
$\{p \mapsto l * ls(l, null)\}$
`let r = ref null in`
$\{p \mapsto l * ls(l, null) * r \mapsto null\}$
$\{p \mapsto l * ls(l, null) * r \mapsto null * ls(null, null)\}$ (symbolic execution rule)
```
while p ≠ null do
  invariant p→_lp ∗ ls(_lp,null) ∗ r→_lr ∗ ls(_lr,null)
```

The second part is to show the loop invariant is preserved by a loop iteration.

```
while !p ≠ null do
  invariant p→_lp ∗ ls(_lp,null) ∗ r→_lr ∗ ls(_lr,null)
```
  $\{p \mapsto l_p * ls(l_p, null) * r \mapsto l_r * ls(l_r, null) * l_p \neq \mathtt{null}\}$
  $\{p \mapsto l_p * l_p \overset{next}{\mapsto} \_q * ls(\_q, null) * r \mapsto l_r * ls(l_r, null) * l_p \neq \mathtt{null}\}$
  `let n = !p→next in`
  $\{p \mapsto l_p * l_p \overset{next}{\mapsto} n * ls(n, null) * r \mapsto l_r * ls(l_r, null) * l_p \neq \mathtt{null}\}$
  `!p→next := !r;`
  $\{p \mapsto l_p * l_p \overset{next}{\mapsto} l_r * ls(n, null) * r \mapsto l_r * ls(l_r, null) * l_p \neq \mathtt{null}\}$
  `r := !p;`
  $\{p \mapsto l_p * l_p \overset{next}{\mapsto} l_r * ls(n, null) * r \mapsto l_p * ls(l_r, null) * l_p \neq \mathtt{null}\}$
  `p := n;`
  $\{p \mapsto n * l_p \overset{next}{\mapsto} l_r * ls(n, null) * r \mapsto l_p * ls(l_r, null) * l_p \neq \mathtt{null}\}$
  $\{p \mapsto n * ls(n, null) * r \mapsto l_p * ls(l_p, null)\}$

Finally, we prove that when we exit the loop, the post-condition is established.

```
while !p ≠ null do
  invariant p→_lp ∗ ls(_lp,null) ∗ r→_lr ∗ ls(_lr,null)
  ...
done
```
  $\{p \mapsto l_p * ls(l_p, null) * r \mapsto l_r * ls(l_r, null) * l_p = \mathtt{null}\}$

$\{p \mapsto null * r \mapsto l_r * ls(l_r, null)\}$
`!r`
$\{ls(result, null)\}$ (implicit dispose!)

6.4 Going Further on Separation Logic

The main motivation of Separation Logic is that it internalizes disjointness and frame properties. It makes reasoning on pointer data structures much easier than a model like the Component-as-Array, where disjointness and frame properties must be stated explicitly.

Separation Logic has several applications and extensions. An important application is to the specification of *data invariants*, that are properties that should remain true along the execution of a program (such as "this list should always contain non-negative integers", "this tree should always be balanced"). Separation Logic provides for free that a data invariant is preserved when memory is modify outside its frame. Separation Logic can also deals with *concurrent programs* [4], for example two threads executing in parallel on disjoint part of the memory can be proved easily.

A major drawback in Separation Logic techniques nowadays is their low level of automation. There no simple equivalent of weakest precondition calculus, and thus SMT solvers cannot be used directly, one has to design provers that can understand the separating conjunction. Indeed, some analogue to WP can be constructed, if one considers also a connective for *separating implication*, also known as *magic wand*[1].

6.5 Exercises

Exercise 6.5.1 *Prove a complete specification of linked list reversal via Separation Logic, using appropriate ghost variables and an extended predicate* `ls` *that includes the model list.*

Exercise 6.5.2 *Specify and prove the function of Exercise 5.3.3 which increments by one each element of a null-terminated linked list of integers, using Separation Logic.*

Exercise 6.5.3 *Specify and prove the function of Exercise 5.3.4 which appends two lists, using Separation Logic.*

Bibliography

[1] J. Berdine, C. Calcagno, and P. W. O'Hearn. Symbolic execution with separation logic. In K. Yi, editor, *APLAS*, volume 3780 of *Lecture Notes in Computer Science*, pages 52–68. Springer, 2005.

[2] J. Berdine, C. Calcagno, and P. W. O'hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *In International Symposium on Formal Methods for Components and Objects*, pages 115–137. Springer, 2005.

[3] A. Charguéraud. Characteristic formulae for the verification of imperative programs. In M. M. T. Chakravarty, Z. Hu, and O. Danvy, editors, *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming (ICFP)*, pages 418–430, Tokyo, Japan, September 2011. ACM.

[4] X. Feng, R. Ferreira, and Z. Shao. On the relationship between concurrent separation logic and assume-guarantee reasoning. In R. D. Nicola, editor, *ESOP*, volume 4421 of *Lecture Notes in Computer Science*, pages 173–188. Springer, 2007.

[1]See `http://www.lsv.ens-cachan.fr/~lozes/sl-lectures.php`

[5] A. Nanevski, G. Morrisett, and L. Birkedal. Polymorphism and separation in Hoare type theory. In J. H. Reppy and J. L. Lawall, editors, *11th ACM SIGPLAN International Conference on Functional Programming, ICFP 2006*, pages 62–73, Portland, Oregon, USA, 2006. ACM Press. ISBN 1-59593-309-3.

[6] J. C. Reynolds. Separation logic: a logic for shared mutable data structures. In *17h Annual IEEE Symposium on Logic in Computer Science*. IEEE Comp. Soc. Press, 2002.

[7] J. Smans, B. Jacobs, F. Piessens, and W. Schulte. An automatic verifier for java-like programs based on dynamic frames. In *Fundamental Approaches to Software Engineering (FASE'08)*, Budapest, Hungary, Apr. 2008.

[8] H. Tuch, G. Klein, and M. Norrish. Types, bytes, and separation logic. In M. Hofmann and M. Felleisen, editors, *Proc. 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'07)*, pages 97–108, Nice, France, Jan. 2007.

Chapter 7

Numeric Programs

Up to this chapter, programs have manipulated numbers that were mathematically ideal: unbounded integers and real numbers. While ideal numbers are useful in logical specifications, they usually do not reflect the computations that happen in a program executed on a practical architecture.

7.1 Handling Machine Integers

Let us consider the set of 32-bit signed integers in 2-complement. It contains all the integers between -2^{31} and $2^{31}-1$ and nothing else. This set is not closed under the usual arithmetic operations, *e.g.* $2^{30} \times 2$ will cause an *overflow*. In absence of an overflow, these operations usually return the mathematical result. In presence of an overflow, the behavior depends on the language and the environment (compiler, architecture, possibly operating system). For instance, the ISO/C language standard mandates that an arithmetic operation on signed integers has no defined behavior in case of overflow.[1] At worst, it might cause the program to crash.

Let us assume that our toy language has the same semantics, or absence thereof. It means that, since behavior is undefined, a program is *safe* only if no overflows occur in arithmetic computations. This constraint can be checked using Hoare logic, assuming all the arithmetic operations that happen in the program (but not in the specification) are replaced by call to functions with preconditions. For instance, the following function is meant to replace the subtraction in programs.

```
function int32_sub(x: int, y: int): int
  requires -2^31 ≤ x - y < 2^31
  ensures result = x - y
```

All the other operators can be handled in a similar way. While sufficient for most program verifications, these function declarations are not complete. Consider the following example that relies on these safe operators.

Example 7.1.1 *Subtracting two nonnegative 32-bit signed integers cannot cause an overflow.*

```
function nonneg_sub(x: int, y: int): int
  requires 0 ≤ x and 0 ≤ y
  ensures result = x - y
  body
    int32_sub(x, y)
```

Yet neither side of the precondition $-2^{31} \leq x - y < 2^{31}$ *can be proved without upper bounds on* x *and* y.

[1]For unsigned integers, the C language mandates that arithmetic operations shall return the integer that is equivalent to the mathematical result modulo 2^{32}. The semantics has no undefined behavior in that case.

A solution is to introduce a new abstract type to represent 32-bit signed integers.

```
type int32
function of_int32(x: int32): int

function int32_sub(x: int32, y: int32): int32
  requires -2^31 ≤ of_int32(x) + of_int32(y) < 2^31
  ensures of_int32(result) = of_int32(x) - of_int32(y)
```

It now becomes possible to assert that all machine integers are bounded.

```
axiom bounded_int32: forall x: int32. -2^31 ≤ of_int32(x) < 2^31
```

This specification is complete, but it requires some additional predicates, since comparison operators from the logic are no longer usable on `int32` values. Equality is no different and should have its own specialized predicate, unless one adds an axiom stating that `of_int32` is injective.

Example 7.1.2 *Example 7.1.1 continued: subtracting two nonnegative 32-bit signed integers cannot cause an overflow.*

```
function nonneg_sub(x: int32, y: int32): int32
  requires 0 ≤ of_int32(x) and 0 ≤ of_int32(y)
  ensures result = of_int32(x) - of_int32(y)
  body
    int32_sub(x, y)
```

Ideally, the logic language should provide implicit coercions from `int32` to mathematical integers so that the use of `of_int32` does not litter specifications, as is the case in Example 7.1.2.

7.2 Floating-Point Computations

On modern computers, floating-point arithmetic is by far the most common way of approximating real arithmetic and hence performing numerical computations. This comes from its inherent mathematical simplicity (huge range and guaranteed precision) mixed with a strong desire of ensuring portability of this arithmetic across all architectures without forsaking performance. These properties usually make this arithmetic an obvious choice for building numerical software.

Floating-point arithmetic is described by the IEEE-754 [1] standard. Basically, a floating-point number represents a real number $m_0.m_1 \ldots m_{p-1} \cdot \beta^e$ where e, called the *exponent*, is an integer, and m, called the *significand*, is represented in radix β. Usual values for the radix are $\beta = 2$ and $\beta = 10$.

In order to handle various exceptional behaviors, a floating-point *datum* can also encode positive and negative infinities, positive and negative zeros, and a special value *Not-a-Number* (NaN for short) which is absorbing.

7.2.1 Definitions and Operations

Formats

Binary floating-point formats are a set of numbers encoded in a 32-, 64-, or 128-bit machine word. These formats are named *binary32*, *binary64*, and *binary128*. The most significant bit is the sign bit s. The next w_e bits represent the nonnegative *biased exponent* e'. The less significant w_m bits are the nonnegative *mantissa* m'. The exponent *bias* is chosen as $2^{w_e-1} - 1$.

The interpretation of these fields depend on the value of the biased exponent e':

- if $e' = 0$, the datum represents the real $(-1)^s \cdot 0.m' \cdot 2^{-bias+1}$,
- if $0 < e' < 2^{w_e} - 1$, the datum represents the real $(-1)^s \cdot 1.m' \cdot 2^{e'-bias}$,
- if $e' = 2^{w_e} - 1$, the datum represents $(-1)^s \cdot \infty$ if $m' = 0$, NaN otherwise.

Floating-point data with $e' = 0$ are called *subnormal* numbers. Signed zeros are subnormal numbers with $m' = 0$. Other numbers are said to be *normal*. The precision p is defined as $w_m + 1$. Decimal floating-point formats follow similar rules, except that the most significant digit of the mantissa is explicit and possibly zero rather than implicit and fixed by the biased exposed.

Below are some special values for the `binary32` and `binary64` formats.

	Precision	Smallest number	Smallest normal number	Biggest number
`binary32`	24	2^{-149}	2^{-126}	$2^{128}(1-2^{-24})$
`binary64`	53	2^{-1074}	2^{-1022}	$2^{1024}(1-2^{-53})$

Lemma 7.2.1 (Representable numbers) *Every real number representable by a floating-point number can be written as $m \cdot 2^e$ with m and e integers satisfying $|m| < 2^p$ and $-bias - p + 2 \le e \le bias - p + 1$. Conversely, every real number that satisfies the constraints above is represented by a single floating-point number, except for zero, which has two associated floating-point numbers.*

Lemma 7.2.2 (Canonical representation) *Any representable real number can be written $m \cdot 2^e$ with either $|m| \ge 2^{p-1}$ or $e = -bias - p + 2$, in addition to the above constraints. This decomposition exists and is unique. Normal numbers satisfy property $|m| \ge 2^{p-1}$.*

Rounding

Representable real numbers are a small subset of the real numbers, of the rational numbers even. Numbers that cannot be represented are *rounded* toward a floating-point datum. There are five traditional rounding directions: to nearest (with tie breaking to even), to nearest (with tie breaking away from zero), downward, upward, toward zero. For binary floating-point arithmetic, the default rounding mode is the first one.

Choosing the rounded value of a real number x can be done as follow. First, select $\underline{x}$, the largest number representable by $m \cdot 2^e$ with m and e integers satisfying $|m| < 2^p$ and $-bias - p + 2 \le e$ that is smaller or equal to x. Notice that there is no upper bound on e. Similarly, select $\overline{x}$, the smallest representable number that is larger or equal to x. Among $\underline{x}$ and $\overline{x}$, choose the number $\tilde{x}$ that satisfies the rounding mode. For instance, when rounding toward zero, the smallest number in absolute value is chosen. If $\tilde{x}$ is too large to be representable in the target format, then an exceptional behavior called *overflow* occurs. Default handling of this exception is to return a floating-point infinity. In case the result is zero, the sign is chosen depending on the rounding mode and the last arithmetic operation.

Lemma 7.2.3 (Local monotonicity) *For any standard rounding mode, any real number between x and its rounded value $\tilde{x}$ is rounded to $\tilde{x}$.*

Proof. By considering all the standard rounding modes in turn.

Floating-point operations

According to the standard, a floating-point operator shall behave as if it was first computing the *infinitely-precise* value and then *rounding* it so that it fits in the destination floating-point format, assuming no exceptional behaviors occur. As a consequence, the result of all the operations can be described in term of rounding. Let rnd a function from $\mathbb{R}$ to $\mathbb{R}$ that, given a real number x, returns its rounded value $\tilde{x}$, irrespective of any possible overflow. In particular, $\mathrm{rnd}(2^e) = 2^e$ for any $e \geq -bias - p + 2$. Function rnd depends on the target format and the rounding direction.

Let `of_binary32` the function from `binary32` datum to real number that returns the represented number. The function is underspecified: the returned value for exceptional floating-point data is left undefined.

Example 7.2.4 *Assuming* rnd *chooses the nearest real representable as a* `binary32` *floating-point number with no upper limit on exponent, the floating-point addition satisfies the following property. Assuming the floating-point sum is a finite number,*

$$\mathit{of\_binary32}(\mathit{binary32\_add}(x, y)) = \mathrm{rnd}(x + y).$$

We say that an operation *underflows* if its infinitely-precise result is smaller than 2^{1-bias} in absolute value. Note that the rounded result might still be in the range of normal numbers, so this behavior is also called *underflow before rounding* in case of ambiguity.

7.2.2 Usual Properties

Floating-point arithmetic does not have many of the properties sought in mathematics, *e.g.* associativity of operators, but it is still a rich structure with numerous properties [9]. Below are a few immediate properties that will be useful as a first step toward specification of numerical programs.

Lemma 7.2.5 (Idempotency) *Given a rounding function* rnd *and a real number* x*,* $\mathrm{rnd}(\mathrm{rnd}(x)) = \mathrm{rnd}(x)$.

Proof. According to the definition of rounding, $y = \mathrm{rnd}(x)$ is either the largest representable number smaller or equal to x or the smallest representable number larger or equal to x. As a consequence, y is a representable number. Similarly $\mathrm{rnd}(y)$ returns either the largest representable number smaller or equal to y or the smallest representable number larger or equal to y. Since y is representable, both numbers are equal to y so $\mathrm{rnd}(y) = y$, which concludes the proof.

As shown in the previous proof, there is no difference between the codomain of rnd and the set $\{m \cdot 2^e \mid |m| < 2^p \wedge e \geq -bias - p + 2\}$. So, from now on, *representable* will indifferently mean any of these properties.

Lemma 7.2.6 (Monotonicity) *Given a rounding function* rnd *and two real numbers* x *and* y*,*

$$x \leq y \Rightarrow \mathrm{rnd}(x) \leq \mathrm{rnd}(y).$$

Proof. First, let us suppose there is a representable number z between x and y. Therefore, we have $\mathrm{rnd}(x) \leq \overline{x} \leq z \leq \underline{y} \leq \mathrm{rnd}(y)$, by definition, and the proof is done. Now let us suppose the opposite. As a consequence, $\underline{x} = \underline{y}$ and $\overline{x} = \overline{y}$. If $\mathrm{rnd}(x) = \underline{x}$, then whatever the value of $\mathrm{rnd}(y)$, $\mathrm{rnd}(x) \leq \mathrm{rnd}(y)$. If $\mathrm{rnd}(x) = \overline{x}$, then $x \leq y \leq \mathrm{rnd}(x)$, which means that $\mathrm{rnd}(y) = \mathrm{rnd}(x)$ by local monotonicity of rounding.

Lemma 7.2.7 (Successor) *Let* m *and* e *two integers such that* $m \cdot 2^e$ *is the canonical representation of a floating-point number. Then* $(m+1) \cdot 2^e$ *is a representable number. Moreover, there is no representable number between* $m \cdot 2^e$ *and* $(m+1) \cdot 2^e$*, when* $m \geq 0$.

Proof. Since $|m| < 2^p$, either $|m+1| < 2^p$ or $m + 1 = 2^p$. In the first case, the number $(m+1) \cdot 2^e$ is immediately representable. In the second case, it is equal to $1 \cdot 2^{e+p}$, which is representable since $e + p \geq e \geq -bias - p + 2$. Any representable number between $m \cdot 2^e$ and $(m+1) \cdot 2^e$ would be written $m' \cdot 2^{e'}$ with $e' < e$ and $m' > m \cdot 2^{e-e'}$, which is impossible given the hypotheses on m and e.

Lemma 7.2.8 (Round-off error without underflow) *Let x a real number and $\tilde{x} = \mathrm{rnd}(x)$ its rounded value. Assuming $|x| \geq 2^{1-bias}$,*

$$\tilde{x} = x \cdot (1 + \varepsilon) \quad \textit{with } |\varepsilon| \leq 2^{-p}$$

for rounding to nearest and $|\varepsilon| < 2^{-p+1}$ for directed rounding.

Proof. Without loss of generality, we can suppose $0 < x$. If x is representable, then $\tilde{x} = x$ and $\varepsilon = 0$ satisfies the properties. Otherwise, let $m \cdot 2^e$ the canonical representation of $\underline{x} \geq 0$. There cannot be any representable number between $\underline{x}$ and $\overline{x}$ and they are different. So, according to Lemma 7.2.7, $\overline{x} = (m + 1) \cdot 2^e$. When rounding to nearest, $|\tilde{x} - x| \leq (\overline{x} - \underline{x})/2 = 2^{e-1} = 2^{p-1+e} \cdot 2^{-p}$. Moreover, by monotonicity, $\underline{x}$ is larger or equal to 2^{1-bias}, so $|m| \geq 2^{p-1}$. As a consequence, $|\tilde{x}/x - 1| \leq 2^{-p}$, which concludes the proof for rounding to nearest. The proof is similar for directed rounding.

Lemma 7.2.9 (Round-off error in case of underflow) *Let x a real number and $\tilde{x} = \mathrm{rnd}(x)$ its rounded value. Assuming $|x| \leq 2^{1-bias}$,*

$$\tilde{x} = x + \delta \quad \textit{with } |\delta| \leq 2^{-bias-p+1}$$

for rounding to nearest and $|\delta| < 2^{-bias-p+2}$ for directed rounding.

Proof. The proof is similar to the previous one, except that, this time, the canonical representation of $\underline{x}$ satisfies $e = -bias - p + 2$.

Combining both lemmas leads to Demmel's model of round-off errors [5].

Lemma 7.2.10 (Round-off error, general case) *Let x a real number and $\tilde{x} = \mathrm{rnd}(x)$ its rounded value.*

$$\tilde{x} = x \cdot (1 + \varepsilon) + \delta \quad \textit{with } |\varepsilon| \leq 2^{-p} \textit{ and } |\delta| \leq 2^{-bias-p+1}$$

for rounding to nearest, and $|\varepsilon| < 2^{-p+1}$ and $|\delta| < 2^{-bias-p+2}$ for directed rounding. Moreover, either $\varepsilon = 0$ or $\delta = 0$.

Lemma 7.2.11 (Round-off error for addition) *Let x and y two representable numbers.*

$$\mathrm{rnd}(x + y) = (x + y) \cdot (1 + \varepsilon) \quad \textit{with } |\varepsilon| \leq 2^{-p}$$

for rounding to nearest and $|\varepsilon| < 2^{-p+1}$ for directed rounding.

Proof. If $|x + y| \geq 2^{1-bias}$, this lemma is a special case of Lemma 7.2.8. Otherwise, let us consider the representations $x = m_x \cdot 2^{e_x}$ and $y = m_y \cdot 2^{e_y}$ with $|m_x|, |m_y| < 2^p$ and $e_x, e_y \geq e$ and $e = -bias - p + 2$. We have $x + y = m \cdot 2^e$ with $m = m_x \cdot 2^{e_x - e} + m_y \cdot 2^{e_y - e}$. Note that m is an integer and $|m| < 2^p$ since $|x + y| < 2^{1-bias}$. As a consequence, $x + y$ is representable, so $\varepsilon = 0$ in that case.

7.2.3 Specifications

Specifications of floating-point arithmetic operators follow the same approach than integer ones. For instance, the following description is suited for verifying a program that is specified not to receive exceptional inputs nor to encounter exceptional floating-point behaviors nor to care about the sign of zero.

```
type binary32
const max_binary32 : real = (2^24 - 1) * 2^102
function of_binary32(x: binary32): real
axiom finite_binary32: forall x: binary32. ???

function rnd...(x: real): real
axiom about_rnd...: ???

function binary32_sub(x: binary32, y: binary32): binary32
  requires abs(rnd...(of_binary32(x) - of_binary32(y))) ≤ max_binary32
  ensures of_binary32(result) = rnd(of_binary32(x) - of_binary32(y))

function binary32_div(x: binary32, y: binary32): binary32
  requires abs(rnd...(of_binary32(x) / of_binary32(y))) ≤ max_binary32
         ∧ of_binary32(y) ≠ 0
  ensures of_binary32(result) = rnd(of_binary32(x) / of_binary32(y))
```

The ellipsis behind rnd reminds that there are several rounding functions. The proper one is selected by the format and the desired rounding mode. Formats and modes could also have been specified as additional arguments to the rounding function. This would be handy to specify a program where the rounding mode is selected dynamically.

The question marks ??? in the axioms are meant to be replaced by statements suitable for the deductive verification of a program. For instance, the characterization of finite floating-point numbers could be taken from Lemma 7.2.1, while the properties of the rounding function could be taken from Section 7.2.2: idempotency, monotonicity, round-off error. Some care should be taken so that these specifications can actually be used by automated provers.

Specifications can also be devised to account for exceptional behaviors by seeing floating-point values as disjoint sums of normal and exceptional values and taking all of the cases into account in post-conditions [2]. Preconditions are then empty since floating-point operations are defined for all inputs.

7.3 Numerical Properties

There are two main kinds of properties one may want to prove about a numerical program. The first kind relates to properties about values of floating-point expressions. They are similar to the ones for integer values, *e.g.* stating that some value is in some domain or that some values are ordered. The other kind of properties relates to the fact that floating-point arithmetic is used in place of real arithmetic, but is only an approximation of it. Basically, verification of numerical programs is concerned with checking that computed values are not too far from ideal mathematical values.

7.3.1 Absolute and Relative Errors

Given two real numbers u and v, the absolute error between u and v usually designates one of the following expressions: $u - v$, $v - u$, or $|u - v|$. Assuming that u is an approximation of v, which is itself an approximation of w, the following properties relate the absolute errors between these three values:

$$\begin{array}{rcccc} u - w & = & (u - v) & + & (v - w) \\ |u - w| & \leq & |u - v| & + & |v - w| \end{array}$$

This "transitivity" is especially useful when it comes to expressions involving rounding functions:

$$|\mathrm{rnd}(u) - v| \leq |\mathrm{rnd}(u) - u| + |u - v|$$

One part can be bounded thanks to Lemma 7.2.10 while the other one no longer involves any rounding function at the top level.

In the case of numerical analysis in general and floating-point arithmetic in particular, the absolute error is not the most natural way to express the quality of an approximation. Indeed, people are usually interested in the number of digits of the result that are meaningful. For this purpose, the relative error (or its logarithm) is a better choice. The relative error between two real numbers u and v designates one of the following expressions: $u/v - 1$, $v/u - 1$, or their absolute values. Again, there are some "transitivity" properties helpful for relating the errors between three values:

$$\frac{u}{w} - 1 = \varepsilon_{uv} + \varepsilon_{vw} + \varepsilon_{uv}\varepsilon_{vw} \quad \text{with } \varepsilon_{uv} = \tfrac{u}{v} - 1 \text{ and } \varepsilon_{vw} = \tfrac{v}{w} - 1$$

7.3.2 Forward Error

Let us consider a mathematical function f and a program $\tilde{f}$ that implements a floating-point computation that supposedly returns a value close to f.

Let us denote u the elementary bound on the relative round-off error assuming no underflow occurs. When rounding to nearest, Lemma 7.2.8 states that $u = 2^{-p}$. In order to ease the handling of round-off errors, let us introduce some notations in the spirit of the mathematical big and small o.

Definition 7.3.1 (γ_n **and** θ_n) *For ε_i such that $|\varepsilon_i| \le u$ and assuming $nu < 1$, we introduce an arbitrary symbol θ_n such that*

$$\prod_{i=1}^{n} (1 + \varepsilon_i)^{\pm 1} = 1 + \theta_n$$

Let $\gamma_n = \frac{nu}{1-nu}$. We have $|\theta_n| \le \gamma_n$.

Example 7.3.2 (Horner scheme) *The following function implements Horner's scheme for computing a floating-point approximation of a polynomial evaluation $\sum_{i=0}^{n} a_i x^i$.*

```
function Horner(a:map int binary32, n:int, x:binary32)
  body
    let y := ref (binary32_cst 0.) in
    let i := ref n in
    while i ≥ 0 do
      y := binary32_add(binary32_mul(y, x), a[i]);
      i := i - 1;
    done;
    y
```

Assuming there are no exceptional behaviors, one can prove that the result computed by the function of Example 7.3.2 is

$$\tilde{f}(x) = \mathrm{rnd}(\ldots \mathrm{rnd}(\mathrm{rnd}(\mathrm{rnd}(a_n x) + a_{n-1})x)\ldots + a_0)$$

Assuming there are no underflows during multiplications, this can be rewritten as

$$\begin{aligned} \tilde{f}(x) &= (\ldots(a_n x(1+\theta_1) + a_{n-1})(1+\theta_1)x(1+\theta_1) + \ldots + a_0)(1+\theta_1) \\ &= a_n x^n (1+\theta_{2n}) + a_{n-1}x^{n-1}(1+\theta_{2n-1}) + \ldots + a_0(1+\theta_1) \end{aligned}$$

As a consequence, the absolute round-off error is bounded by

$$
\begin{aligned}
|\tilde{f}(x) - f(x)| &= |a_n x^n \theta_{2n} + \ldots a_0 \theta_1| \\
&\leq |a_n x^n| \gamma_{2n} + \ldots + |a_0| \gamma_1 \\
&\leq \gamma_{2n} \sum_{i=0}^{n} |a_i| \cdot |x|^i
\end{aligned}
$$

Note that the quotient $(\sum_{i=0}^{n} |a_i| \cdot |x|^i)/|f(x)|$ tends to infinity when x is near a root of the polynomial. As a consequence, the relative round-off error might be unbounded in that case (and it is in practice).

7.3.3 Backward Error

The forward error expresses the accuracy of the result, but it is not the only property one can be interested in proving about a numerical program. Another one is the backward error, though it is often encountered as a stepping stone to obtain properties about the forward error [7].

The backward error of $\tilde{f}(x)$ is the distance between x and the closest real number $\tilde{x}$ such that $\tilde{f}(x) = f(\tilde{x})$, assuming it exists. In other words, rather than considering that $\tilde{f}(x)$ is an approximated result of $f(x)$, backward error analysis assumes it is the exact result of f evaluated at some approximated input $\tilde{x}$.

Example 7.3.3 (Backward analysis of the 2×2-determinant) *Consider the floating-point evaluation of a determinant* $\widetilde{\det}(a, b, c, d) = \mathrm{rnd}(\mathrm{rnd}(ad) - \mathrm{rnd}(bc))$ *and assume the evaluation causes no underflow nor overflow.*

$$
\begin{aligned}
\widetilde{\det}(a, b, c, d) &= (ad(1 + \theta_1) - bc(1 + \theta_1))(1 + \theta_1) \\
&= a(1 + \theta_1) \times d(1 + \theta_1) - b(1 + \theta_1) \times c(1 + \theta_1) \\
&= \tilde{a}\tilde{d} - \tilde{b}\tilde{c} \quad \textit{with } \tilde{a} = a(1 + \theta_1) \textit{ and so on} \\
&= \det(\tilde{a}, \tilde{b}, \tilde{c}, \tilde{d})
\end{aligned}
$$

So the relative backward error of $\widetilde{\det}$ *for the* $\|\cdot\|_\infty$ *norm is bounded by* γ_1. *In other words, computing the approximation* $\widetilde{\det}$ *is no different than computing the exact* $\det$ *on slightly perturbed inputs.*

7.4 Automatizing the Proof of Numerical Algorithms

Assuming that the specifications of a program only mention the properties described in the previous section, that is, properties on values and bounds on errors, the approach to verifying it depends on the structure of the program. We will first consider the case where the program matches the mathematically ideal algorithm, that is, executing the program with an infinitely-precise arithmetic gives the expected result and the program contains no superfluous operations.

7.4.1 Ghost Values

To ease the automated verification of numerical programs, we will now consider that floating-point values are more than that. In addition to the value computed by a floating-point operation, they will also carry the real number that would have been computed with an infinitely-precise arithmetic. This can be expressed by using ghost variables, as was done in Section 5.2.4.

For instance, the 2×2-determinant function could be written as

```
function det(a b c d: binary32, aM bM cM dM: real): (binary32, real)
  body
    let t1 := binary32_mul(a, d) in
    let t1M := aM * dM in
    let t2 := binary32_mul(b, c) in
    let t2M := bM * cM in
    let t3 := binary32_sub(t1, t2) in
    let t3M := t1M - t2M in
    (t3, t3M)
```

The forward error of the algorithm would then be a property about `t3 - t3M` or `t3 / t3M - 1`.

7.4.2 Abstract Interpretation

The framework of abstract interpretation provides some useful tools to automatize the verification. The main difference with the traditional approach of abstract interpretation is that it will be used not only on the computed values, but also on the ghost/model values and on their relations [6].

With abstract interpretation, all the ghost values are implicit and only the actual computed values are taken into account. A floating-point variable will therefore represent a pair (computed, model). The specifications of floating-point operations are easily extended to support such a model [3]:

```
function of_binary32(x: binary32): real
function model_of_binary32(x: binary32): real

function binary32_add(x: binary32, y: binary32): binary32
  requires abs(rnd...(of_binary32(x) + of_binary32(y))) ≤ max_binary32
  ensures of_binary32(result) = rnd(of_binary32(x) + of_binary32(y)) ∧
    model_of_binary32(result) = model_of_binary32(x) + model_of_binary32(y)
```

Then domains can be devised to abstract the computed values, the model values, and more importantly, the error between them. For instance, a simple domain could store the computed values as intervals, the errors as a bound on their absolute value, and it could discard the model values. These would give the following rule for addition. Given some enclosures $[\underline{x}, \overline{x}]$ of x, $[\underline{y}, \overline{y}]$ of y, and some error bounds δ_x and δ_y such that $|x - x_M| \le \delta_x$ and $|y - y_M| \le \delta_y$, the floating-point sum z of x and y would satisfy:

$$z \in [\mathrm{rnd}(\underline{x} + \underline{y}), \mathrm{rnd}(\overline{x} + \overline{y})] \quad \text{and} \quad \delta_z = \delta_x + \delta_y + \gamma_1 \max(\overline{x} + \overline{y}, -(\underline{x} + \underline{y}))$$

Note that, if the lower, upper, and error bounds in input, are numerical values, then the rule produces numerical values too. Similar rules can be designed for all the floating-point operations.

7.4.3 Round-off and Method Errors

Some mathematical expressions cannot be represented as formulas suitable for numerical programs. For instance, they may depend on an infinite amount of operations, or on operations not available in the target language/architecture, or they may simply describe a result implicitly by a (differential) equation.

In that case, the usual way to verify the correctness of an algorithm is to split the analysis of the error in two parts: the round-off error, which is the distance between the computed valued and the infinitely-precise value, and the *method error*, which is the distance between the infinitely-precision expression and the original expression. The round-off error can be handled by the previous approach, while the method error is pure problem of numerical analysis without any floating-point number. Both errors can then be merged as was explained in Section 7.3.1.

The method error is also called the *truncation* error when the programs just evaluate partial sums or products, while the ideal expressions contain infinite sums or products.

7.5 A Classification of Numerical Algorithms

For efficiency and/or accuracy reasons, some numerical algorithms are not a straightforward translation of mathematical formulas to floating-point arithmetic. As a consequence, they do not fit the previous framework and their correctness proof requires some human intervention. Three categories of algorithms are presented below. They are not meant to be exhaustive, but rather to present some of the difficulties one might encounter when verifying a numerical algorithm. Each of them is illustrated by an example.

1. The programs depend on some mathematical property that is not readily available from the code to ensure their accuracy.
2. The programs perform a lot of extraneous computations, which, if they were performed with an infinitely-precise arithmetic, would just be equal to zero. Yet they are not just noise and they greatly increase the accuracy of the algorithm.
3. The programs, if they were performed with an infinitely-precise arithmetic, would just return nonsensical results. So their correctness cannot be proved by comparing them to their infinitely-precise counterparts.

From now on, program examples will no longer use explicit functions for floating-point arithmetic. For the sake of readability, they will just use mathematical operators in code. Keep in mind that each of them involves a rounding. In specifications, however, mathematical operations still denote the infinitely-precise operators on real numbers.

7.5.1 Converging Algorithms

These programs are still a straightforward approximation of the same algorithms on real numbers, but their correctness actually depends on some mathematical property, *e.g.* convergence. In particular, studying separately the method error and the round-off error of the algorithms can make their verification harder, if not impossible.

Example 7.5.1 (Square root computed by Newton's iteration) *The following program is supposed to compute a rather accurate approximation of the square root of an input x between 0.5 and 2. The first function computes a poor approximation of the inverse square root. Its code is not shown; it could be a small polynomial interpolation or just some table lookup.*

The second function, starting from the result of the first function, performs three iterations of Newton's iteration in order to refine the approximation of the inverse square root until it is accurate enough. Finally it multiplies it by x to get an approximation of $\sqrt{x}$.

```
function fp_sqrt_init(x:binary64) : binary64
  requires 0.5 ≤ x ≤ 2;
  ensures abs(result - 1/sqrt(x)) ≤ 2^-6 * 1/sqrt(x);

function fp_sqrt(x:binary64) : binary64
  requires 0.5 ≤ x ≤ 2;
  ensures abs(result - sqrt(x)) ≤ 2^-43 * sqrt(x);
  body
    let t := ref (fp_sqrt_init(x)) in
let i := ref 0 in
while i < 3 do
  t := 0.5 * t * (3 - t * t * x);
  i := i + 1;
done;
t * x
```

First of all, if one were to apply the method from the previous section, one would first bound the round-off error between `fp_sqrt` and a bivariate polynomial on `x` and `fp_sqrt_init(x)`, then bound the method error between this polynomial and $\sqrt{x}$, and finally combine both errors to get the accuracy of the `fp_sqrt` function. This approach does not work in practice: it gives a poor bound on the error unless a lot of time is wasted on the verification.

The behavior of the above example is best understood if one notices that the following equality holds for any u:

$$0.5u(3 - u^2x)\sqrt{x} - 1 = -(1.5 + 0.5(u\sqrt{x} - 1)) \times (u\sqrt{x} - 1)^2 \qquad (7.1)$$

If we apply the previous equality to the iteratively computed values of t (noted t_0, t_1, and so on). We get

$$t_{n+1}\sqrt{x} - 1 \simeq 0.5t_n(3 - t_n^2x)\sqrt{x} - 1 \simeq -1.5(t_n\sqrt{x} - 1)^2$$

The value $t_n\sqrt{x} - 1$ is the relative error between t_n and $1/\sqrt{x}$. As can be seen from the previous formula, this value is (almost) squared at each iteration, which means that if t_n correctly approximates the first k bits of $1/\sqrt{x}$, then t_{n+1} will approximate almost $2k$ bits of $1/\sqrt{x}$. Computations are performed by floating-point arithmetic, so some round-off errors have to be taken into account too. Note that the round-off error introduced at a given iteration partly vanishes during the next iteration due to the *quadratic convergence*. As a consequence, one can have the intuition that starting from t_0 accurate to 6 bits, t_1 will be accurate to almost 12 bits, t_2 to 23 bits, and t_3 to 45 bits.

This property can be expressed as a loop invariant about `abs(t * sqrt(x) - 1)` and `i`, which guarantees the correctness of the `fp_sqrt` function once verified. The proof goes as follows:

1. deduce from the enclosures of $|t\sqrt{x} - 1|$ (invariant) and x (precondition) the domain of t;
2. perform forward error analysis to bound the relative round-off error between the next value of t and $0.5t(3 - t^2x)$;
3. prove Equation 7.1 and use it to bound the relative method error between $0.5t(3-t^2x)$ and $1/\sqrt{x}$;
4. combine both previous points to deduce that the invariant still holds after a loop iteration.

The fact that the round-off error vanishes at each iteration is what a separate analysis of the round-off error and the method error would miss.

7.5.2 Compensated Algorithms

These programs perform seemingly useless operations in order to improve the accuracy of their results.

Example 7.5.2 (Accurate summation [11]) *The following program takes as inputs an array `x` of length `n` and returns in `s'` an accurate sum of its elements.*

```
s := x[0];
e := 0.;
for i := 1 to n - 1 do
  y := x[i];
```

```
    t := s + y;
    u := t - y;
    r := (s - u) + (y - (t - u));
    s := t;
    e := e + r;
  done;
  s' := s + e;
```

Notice that the value of s is just the naive floating-point sum of all the values of the array x. That means that, if s' actually computes an accurate sum, then the value of e has to *compensate* most of the round-off errors that were accumulated during the computation of s in the loop.

First, let us see what the round-off error for s is.

$$\begin{aligned} s &= (\ldots((x_0 + x_1)(1+\theta_1) + x_2)(1+\theta_1) + \ldots + x_{n-1})(1+\theta_1) \\ &= x_0(1+\theta_{n-1}) + x_1(1+\theta_{n-1}) + \ldots + x_{n-1}(1+\theta_1) \\ &= \sum x_i(1+\theta_{n-1}) \end{aligned}$$

Therefore, $|s - \sum_i x_i| \leq \gamma_{n-1} \sum_i |x_i|$. In other words, s may be a poor approximation of $\sum_i x_i$ if $|\sum_i x_i|$ is much smaller than $\sum_i |x_i|$.

To understand how e ends up accurately representing the accumulated round-off error of s, consider the following lemmas.

Lemma 7.5.3 (Sterbenz) *Let x and y two representable numbers. If $x/2 \leq y \leq 2x$, then $\mathrm{rnd}(x-y) = x - y$.*

Lemma 7.5.4 (Representable error for addition) *Let x and y two representable numbers. The real number $\mathrm{rnd}(x+y) - (x+y)$ is representable by a floating-point number.*

Lemma 7.5.5 (Dekker's error-free addition [4]) *Let x and y two representable numbers. Let $s = \mathrm{rnd}(x+y)$ and $e = \mathrm{rnd}(y - \mathrm{rnd}(s-x))$ two floating-point values computed by rounding to nearest. If $|x| \geq |y|$, then $s + e = x + y$..*

Proof. The sketch of the proof is as follows. First, s has the same sign as x. Without loss of generality, let us assume they are nonnegative. We have $s \leq 2x$. If $s = x + y$ exactly, then all the intermediate values are representable, so $e = 0$ and $s + e = x + y$. Otherwise, the contrapositive of Lemma 7.5.3 ensures that $-y < x/2$. So $x/2 < x + y$, which means that s and x satisfies the hypothesis of Lemma 7.5.3. As a consequence, $\mathrm{rnd}(s - x) = s - x$. Since the round-off error of addition is representable,

$$e = \mathrm{rnd}(y - \mathrm{rnd}(s-x)) = \mathrm{rnd}(y - (s-x)) = (x+y) - s$$

Lemma 7.5.6 (Knuth' error-free addition [8]) *Let x and y two representable numbers. Assuming the following values are computed with floating-point arithmetic rounded to nearest*

```
s := x + y;
u := s - y;
e := (x - u) + (y - (s - u));
```

then $s + e = x + y$.

Applying the previous lemma to Example 7.5.2, one gets that the value of e at the end of the algorithm is the naive floating-point sum of all the round-off errors that happened during the computation of s in the body of the loop. In other words, if the addition at line 9 was performed with an infinitely-precise arithmetic, then we would have $s + e = \sum_i x_i$ at the end of the loop. Due to the floating-point operation at line 9, the error is

$$\left|s' - \sum x_i\right| \leq \gamma_1 \left|\sum x_i\right| + \gamma_{n-1}^2 \sum |x_i|$$

Notice that, compared to the naive sum, the absolute error that depends on $\sum |x_i|$ has been multiplied by γ_{n-1} and is therefore much smaller. The accuracy of this algorithm can be further improved by adding some other floating-point operations to accurately sum e.

7.5.3 Magical Algorithms

Example 7.5.7 (Payne and Hanek's argument reduction [10]) *Consider the following function. Given a large input x, it returns a pair (y, k) such that $0 \leq y \lesssim \pi/4$ and $y + k\pi/4$ is equivalent to x modulo 2π. In other words, it returns values suitable for computing a trigonometric function. Note that all the intermediate computations are performed in* binary64 *format.*

```
function reduce(x:binary32): (binary32, int)
  requires 2^31 ≤ x < 2^32
  ensures exists l:int. abs((result + k * pi/4) - (x + l * 2*pi)) ≤ 2^-25
  body
    let x' := binary64_of_binary32 x in
    let t := x' * 0.02323954474... in
    let k := trunc(t) in
    let y := (t - k) * 0.785398163... in
    (binary32_of_binary64(y), k)
```

The purpose of the above function is to compute x modulo $\pi/4$. So, ideally, it should strive to approximate

$$y \simeq \frac{\pi}{4}\left(\frac{4}{\pi}x - \left\lfloor \frac{4}{\pi}x \right\rfloor\right)$$

While one can recognize in the program above that $0.785\ldots$ stands for $\pi/4$, the constant $0.0232\ldots$ is not $4/\pi$ at all. So why does the program even works? Moreover, how come that the program no longer works if one replaces $0.0232\ldots$ by an approximation of $4/\pi$?

Let us write $4/\pi$ as $(\alpha_h + \alpha_l)2^{-5}$ with α_h an integer and $0 \leq \alpha_l < 1$. The value represented by x can be written $m \cdot 2^8$ with m an integer since x is a binary32 number larger than 2^{31}. As a consequence,

$$\frac{4}{\pi}x = 8\alpha_h m + \alpha_l x 2^{-5}$$

Since $\alpha_h m$ is an integer, $(\pi/4)8\alpha_h m = 2\pi\alpha_h m$ has no impact when considered modulo 2π. Therefore, rather than using the constant $4/\pi \simeq 1.273\ldots$, one can use the constant $C = 4/\pi - \alpha_h 2^{-5} \simeq 0.0232\ldots$.

Why is the result y more accurate with the new constant? First, one can assume that the difference $t - k$ can be represented exactly, since either k is zero or Lemma 7.5.3 applies ($k \leq t < k + 1 \leq 2k$). As a consequence,

$$y = (t - \lfloor t \rfloor)(\pi/4(1 + \theta_1))(1 + \theta_1)$$

For the sake of simplicity, let us assume that $\lfloor t \rfloor = \lfloor Cx \rfloor$. Therefore,

$$\begin{aligned} y &= (x(C(1 + \theta_1))(1 + \theta_1) - \lfloor Cx \rfloor)(\pi/4(1 + \theta_1))(1 + \theta_1) \\ &= (Cx - \lfloor Cx \rfloor)\pi/4(1 + \theta_2) + xC\pi/4\theta_2(1 + \theta_2) \end{aligned}$$

The left-hand side is the ideal mathematical value with some negligible relative error. The right-hand side, however, is far from negligible. So the smaller C is, the more accurate the argument reduction is. For $|x| < 2^{32}$ and $C \simeq 0.0232$, this gives an absolute error of about 2^{-25}.

Note that only part of the full algorithm is presented here. For each slice $[2^k, 2^{k+1})$ of input x, a different constant C_k should be chosen, otherwise the absolute error will grow proportionally to x. The storage for the constants (C_k) is hardly an issue, since they share most of their bits. Indeed, to get from C_k to C_{k+1}, the most significant bit is discarded and a bit of the binary expansion of π is added as the least significant bit. In other words, one only needs to know about one hundred bits of the binary expansion of π and perform a bit of bit-fiddling to get all the constants. Note that C_k varies like 2^{26-k}, so the absolute error stays around 2^{-25} whatever the range of x, which would not be the case if one were to use $4/\pi$ as a constant.

7.6 Exercises

Exercise 7.6.1 *Propose a way to support numeric constants of type `int32` inside programs.*

Exercise 7.6.2 *Propose a specification for saturating arithmetic operations on signed integers.*

Exercise 7.6.3 *Prove that a square root computation can cause neither underflow nor overflow. Deduce a simple specification for a floating-point square root that does not depend on a rounding function.*

Exercise 7.6.4 *Suggest some preconditions on x, n, and $(a_i)_{0\le i\le n}$, such that the evaluation of Horner's scheme in Example 7.3.2 does not encounter any underflow.*

Exercise 7.6.5 *How should the code be modified in Example 7.5.1 so that it computes the square root accurately (for the relative error) not just on* $[0.5; 2]$ *but on the whole nonnegative floating-point range.*

Bibliography

[1] IEEE standard for floating-point arithmetic. Technical report, 2008.

[2] A. Ayad and C. Marché. Multi-prover verification of floating-point programs. In J. Giesl and R. Hähnle, editors, *Fifth International Joint Conference on Automated Reasoning*, volume 6173 of *Lecture Notes in Artificial Intelligence*, pages 127–141, Edinburgh, Scotland, July 2010. Springer. URL `http://www.lri.fr/~marche/ayad10ijcar.pdf`.

[3] S. Boldo and J.-C. Filliâtre. Formal Verification of Floating-Point Programs. In *18th IEEE International Symposium on Computer Arithmetic*, pages 187–194, Montpellier, France, June 2007. URL `http://www.lri.fr/~filliatr/ftp/publis/caduceus-floats.pdf`.

[4] T. J. Dekker. A floating point technique for extending the available precision. *Numerische Mathematik*, 18(3):224–242, 1971.

[5] J. Demmel. Underflow and the reliability of numerical software. *SIAM Journal on Scientific and Statistical Computing*, 5(4):887–919, 1984.

[6] E. Goubault and S. Putot. Static analysis of numerical algorithms. In K. Yi, editor, *SAS*, volume 4134 of *LNCS*, pages 18–34. Springer, 2006. ISBN 3-540-37756-5.

[7] N. J. Higham. *Accuracy and stability of numerical algorithms*. SIAM, 2002. URL `http://www.ma.man.ac.uk/~higham/asna/`.

[8] D. E. Knuth. *The Art of Computer Programming, volume 2 (3rd ed.): Seminumerical Algorithms*. Addison-Wesley Longman Publishing Co., Inc., 1997.

[9] J.-M. Muller, N. Brisebarre, F. de Dinechin, C.-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, and S. Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser, 2010.

[10] M. Payne and R. Hanek. Radian reduction for trigonometric functions. *SIGNUM Newsletter*, 18: 19–24, 1983.

[11] S. M. Rump, T. Ogita, and S. Oishi. Accurate floating-point summation part I: Faithful rounding. *SIAM Journal on Scientific Computing*, 31(1):189–224, 2008. doi:10.1137/050645671.