Iterators: where folds fail

Sylvie Boldo*†
Inria, Université Paris-Saclay, F-91893 Palaiseau
LRI, CNRS & Univ. Paris-Sud, F-91405 Orsay

This paper is about the formal verification of a floating-point case study where the common iterators `fold_left` and `fold_right` have not the wanted behaviors. We then had to define other iterators, which are very similar in most cases, but that do behave well in our case study.

Our case study: floating-point expansions

This is about floating-point arithmetic, defined by the IEEE-754 standard [7, 4]. It has been formalized in the Coq proof assistant by the Flocq library [2].

Our case study is a way to increase the precision of a computation, in order to get more accuracy on critical systems. To retain efficiency, we rely on the floating-point unit (FPU) of the processor. The idea of these floating-point expansions is to consider x as $x = \sum x_i$ with the x_i being floating-point numbers, see Figure 1. Using exact addition and multiplication [3, 6], we can build correct operations on these expansions such as addition, multiplication, division, and so on [3, 8, 9, 1].

We are in the progress of proving operations on expansions. It is important as there may be intermediate zeroes in these expansions that may endanger the algorithms. Moreover, the condition of use of these algorithm: allowed overlap, underflow, overflow, and so on are hardly known, and not formally verified.

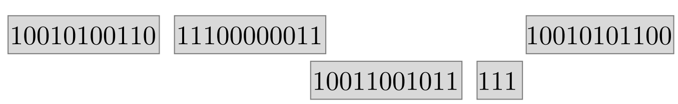

Figure 1: Example of a floating-point expansion.

*This work was supported by the FastRelax (ANR-14-CE25-0018-01) project of the French National Agency for Research (ANR).
†Email: `sylvie.boldo@inria.fr`

Iterators

Here is the Coq definition of `fold_left`:

```
Fixpoint fold_right (l:list B) : A :=
    match l with
      | nil => a0
      | cons b t => f b (fold_right t)
    end.
```

In particular, it relies on the definition of `a0:A` that stands for the zero of `A`. For real numbers, we can consider `fold_right Rplus R0 (1::2::3::nil)` that will be simplified in `1 + (2 + (3 + 0))`. This is perfect as `Rplus` is the mathematical $+$ on $\mathbb{R}$ with `R0` $= 0$ is the identity element. If you consider `f` as the floating-point addition, it is fine as 0 is also an identity element for this function.

Now consider the function $errf(x, y) = \circ(x + y) - (x + y)$, with $\circ$ being a rounding to nearest. Then f computes the error of the floating-point addition, which is also a floating-point number when x and y are floating-point numbers, and that may be computed using only floating-point operations [3, 6]. Then `fold_right errf R0 (a::b::c::nil)` is `(errf a (errf b (errf c 0)))`. As $errf(x, 0) = 0$, this value is 0 whatever a, b and c. And this is not what we want as we wished to have `(errf a (errf b c))`. This is not a toy example as such an iteration of $errf$ on a sequence is the first part of a renormalization algorithm [5].

Solution

Of course, this problem with `fold_right` also applies to `fold_left`. We have therefore defined two other iterators:

```
Definition my_iter_l f l :=
   match l with
   | nil => 0
   | x :: l => fold_left f l x
   end.

Definition my_iter_r f l :=
  match l with
   | nil => 0
   | _ => fold_right f (last l 0) (removelast l)
   end.
```

They have the expected behavior on the previous example. The correctness of `my_iter_r` is given by the following theorem:

```
Lemma my_iter_r_fold_right: forall (f:R->R->R) b l,
   (forall x, f x b = x) -> (l <> nil \/ b = 0) ->
    my_iter_r f l = fold_right f b l.
```

If there exists an identity element b, then the two iterators have the same result, providing either $b = 0$ or the list is not empty. A similar result for `my_iter_l` is useless as it simplifies immediately on `fold_left` as soon as the list is non empty. Several useful lemmas have also been proved, such as `my_iter_l f l = my_iter_r (fun x y =>f y x) (rev l)`.

As a conclusion, we have defined other iterators that comply with our needs. In particular, they do not expect an identity element on the function, but are more difficult to handle in the proofs. They could have been made more generic, in the returned value when the list is empty (here 0), and in the input type (here $\mathbb{R}$). It is nevertheless surprising that this problem has not arisen before on other applications.

References

[1] David H. Bailey. A Portable High Performance Multiprecision Package. Technical Report RNR-90-022, NASA Ames Research Center, Moffett Field, California, May 1993.

[2] Sylvie Boldo and Guillaume Melquiond. Flocq: A unified library for proving floating-point algorithms in Coq. In Elisardo Antelo, David Hough, and Paolo Ienne, editors, *20th IEEE Symposium on Computer Arithmetic*, pages 243–252, Tübingen, Germany, 2011.

[3] T. J. Dekker. A Floating-Point Technique for Extending the Available Precision. *Numerische Mathematik*, 18:224–242, 1971.

[4] David Goldberg. What every computer scientist should know about fl oating point arithmetic. *ACM Computing Surveys*, 23(1):5–47, 1991.

[5] Mioara Joldes, Olivier Marty, Jean-Michel Muller, and Valentina Popescu. Arithmetic algorithms for extended precision using floating-point expansions. *IEEE Transactions on Computers*, 65(4):1197 – 1210, April 2016.

[6] Donald Ervin Knuth. *The Art of Computer Programming: Seminumerical Algorithms*, volume 2. Addison Wesley, Reading, Massachusetts, second edition, 1981.

[7] Microprocessor Standards Committee. IEEE Standard for Floating-Point Arithmetic. *IEEE Std. 754-2008*, pages 1–58, August 2008.

[8] Douglas M. Priest. Algorithms for arbitrary precision floating point arithmetic. In Peter Kornerup and David Matula, editors, *Proceedings of the 10th Symposium on Computer Arithmetic*, pages 132–144, Grenoble, France, 1991. IEEE Computer Society Press.

[9] Jonathan Richard Shewchuk. Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates. *Discrete & Computational Geometry*, 18(3):305–363, October 1997.