Adjustable References

Viktor Vafeiadis

Max Planck Institute for Software Systems (MPI-SWS)

Abstract. Even when programming purely mathematical functions, mutable state is often necessary to achieve good performance, as it underlies important optimisations such as path compression in union-find algorithms and memoization. Nevertheless, verified programs rarely use mutable state because of its substantial verification cost: one must either commit to a deep embedding or follow a monadic style of programming. To avoid this cost, we propose using adjustable state instead. More concretely, we extend Coq with a type of adjustable references, which are like ML references, except that the stored values are only partially observable and updatable only to values that are observationally indistinguishable from the old ones.

1 Introduction

Interactive proof assistants (Coq [3], Isabelle [13], HOL [11], etc.) are generally very good at reasoning about terminating purely functional computations, as these are just mathematical functions, which the provers support natively. In contrast, however, they are not so good at reasoning about real computations, which are not necessarily always terminating or purely functional. Such computations are not supported natively, but have to be encoded in some non-trivial way. This is a problem, because while requiring termination or productiveness for real computations may seem a reasonable restriction, disallowing mutable state is not at all reasonable, as many programs use mutable state for efficiency.

There are essentially two approaches for encoding stateful computations in an interactive proof assistant, both of which have their limitations.

First, in the *deep embedding approach* (e.g., [1,4,14]), impure computations are represented as terms of a custom data structure. Then, to reason about such terms, the user effectively has to build a specialized theorem prover for such terms. This is at the same time a rather challenging and a rather mundane task, as one cannot reuse much of the infrastructure of the interactive theorem prover, but may rather have to (re)implement standard features such as variable binding (cf. the PoplMark challenge [2]) or equating terms up to reduction.

Alternatively, there is the *monadic approach* (e.g., [8,9]), where imperative code is written in a monadic style against the state monad. This approach is usually preferable to the deep embedding approach as it reuses the interactive prover's infrastructure. It is nevertheless problematic because one cannot use global state to optimise purely functional code without forcing all the code depending on it to be written in a monadic style. Consider, for example, a mathematical function of type $A \to B$. If we apply the memoization optimisation (remembering the results of earlier function invocations in a hashtable so as to avoid recomputation), we will get a function of type $A \to \mathsf{StateMonad}\, B$. Thus, the memoized function, although intuitively equivalent to the original one, has a different type and can, therefore, no longer be used in arbitrary contexts, but only within contexts expecting monadic stateful computations as arguments.

To avoid these problems, we introduce a restricted form of mutable state, which we call *adjustable references*, that can be used freely inside pure computations (see §2). Similar to mutable references in ML, adjustable references store some internal value, but unlike ML references, the values stored cannot be arbitrarily updated. They can only be 'adjusted' so that the represented value remains the same, but perhaps affecting the cost of returning/computing it. From a verification point of view, we have to show that every adjustment is effectively an identity function. Then, when reasoning about code using adjustments, we can simply ignore the adjustments. In the extracted implementations, however, we perform the adjustments, as these can be very beneficial in terms of performance.

Adjustable references allow us to implement advanced persistent data structures by enabling imperative updates provided that they affect only the efficiency of the data structure accessor methods and not their results. As examples, we present two standard imperative optimisations that can be easily expressed using adjustable references: (*i*) memoization of function calls (see §3) and (*ii*) path compression of the union-find representation tree (see §4). The formal development associated with this paper can be found at the URL below:

`http://www.mpi-sws.org/~viktor/arefs/`

2 Adjustable References

In this section, we present an axiomatisation of adjustable references in Coq [3] and two implementations: (*i*) a purely-functional one in Coq that ensures the logical consistency of the axioms, and (*ii*) an efficient imperative one in OCaml. While adjustable references can be encoded in other proof assistants, we remark that our axiomatisation uses dependent types.

Our axiomatisation uses the Coq **Axiom** `x : T` and **Parameter** `x : T` declarations, which extend the context axiomatically with a new global constant, `x`, having the given type, `T`. (The former is typically used for propositions, and the latter for other types.)

Adjustable references internally store values of a representation type, R, but do not allow direct read-access to those values. The values are instead observable only at a possibly different type, T. Each adjustable reference type, `aref` f, thus has an associated observation function, $f : R \to T$, mapping values of its internal representation type to ones of its observation type.

```
Parameter aref : ∀R T, (R → T) → Type.
```

Adjustable references have a canonical constructor, `aref_val` $f\,v$, which creates a new reference of type `aref` f holding the internal value v. Our first axiom insists that every adjustable reference contains some internal value.

```
Parameter aref_val : ∀R T (f: R → T), R → aref f.
Axiom aref_inh :
    ∀ R T (f: R → T) (r: aref f), ∃v, r = aref_val f v.
```

Further, we have an operation, `aref_get` r, which reads the adjustable reference r and returns its 'external' value—namely, the result of applying the observation function to its internal value.

```
Parameter aref_get : ∀R T (f: R → T), aref f → T.
Axiom aref_get_val :
    ∀ R T (f: R → T) v, aref_get (aref_val f v) = f v.
```

Next, we need a way of adjusting the contents of the reference cell. Naively, one might think of axiomatising an operation of the following type:

$$\forall R, T.\ \forall f : R \to T.\ \forall r : \texttt{aref}\,f.\ \forall v : R.\ f(v) = \texttt{aref\_get}\,(r) \to \mathsf{unit}\,.$$

That is, we should be able to replace the r's current internal value with any value, v, that is observationally indistinguishable: i.e., $f(v) = \texttt{aref\_get}\,(r)$.

There are, however, three problems with this type.

1. First, Coq does not fix an evaluation order. In particular, the evaluation of $e_1; e_2$ (standing for **let** $x = e_1$ **in** e_2 where $x \notin \mathsf{fv}(e_2)$) may run e_1 and e_2 in any order. Moreover, it may not even evaluate e_1 at all since its value is never used. Coq's extraction [7], for example, does exactly so. It transforms $e_1; e_2$ into e_2, thereby erasing any 'adjustments' made in e_1. This problem on its own is not insurmountable, as we can force the evaluation of e_1 by redefining $e_1; e_2$ to mean **match** e_1 **with** $\mathsf{tt} \Rightarrow e_2$ **end**.
2. Second, the type above is overly restrictive. It requires the new internal value to be provided directly, thereby allowing it to depend only on the observable values of adjustable reference cells. In particular, it cannot depend on the previous internal value of the same reference cell. This is problematic if we want to use adjustable references to store some sort of cache. A typical adjustment of such a reference would be to extend the cache with one more entry, but to do so, the adjusted cache should clearly be able to depend on the old cache.
3. Third, unless the externally observable type, T, is a function type, $A \to B$, there is not much point in using adjustable references: we can just as simply calculate $f(v)$ at creation time, and then store the result in an immutable reference. For a function type, however, one may gradually refine the stored internal value each time the function is called, and therefore potentially improve the performance of future function calls.

For these reasons, we provide a combined adjustment read operation customized to the case of $T = (A \to B)$.

```
Parameter aref_getu :
  ∀ R A B (f : R → A → B) (upd : R → A → R * B)
      (PF: ∀x a, f (fst (upd x a)) = f x)
      (PF': ∀x a, snd (upd x a) = f x a),
    aref f → A → B.
```

Here, we must provide a function, upd, which when given the current internal representation and the function argument returns a pair consisting of the new representation and the result of calling f with these two arguments.

Adjusting reads have a remarkably simple axiomatisation: logically they are exactly the same as normal reads!

```
Axiom aref_getuE : ∀R A B (f : R → A → B) upd PF PF',
                     aref_getu upd PF PF' = aref_get (f:= f).
```

Coq experts will note that the proof obligations of `aref_getu` use the normal propositional Leibniz equality, and may be concerned that these proof obligations will be difficult to satisfy if B is itself a function type. This apparent problem, however, is not very serious as there are multiple ways around it. For example, one may assume the functional extensionality axiom when doing such proofs, as the axiom is consistent with Coq and its use will moreover not impact execution, since extraction erases these arguments. Alternatively, one may uncurry the function type, T, or simply extend the definition to one expecting a curried function of n arguments; i.e., with $T = (A_1 \to \ldots \to A_n \to B)$.

When adjusting the value of a reference cell, we have seen that it is useful for the new internal value to depend on the old internal value of the cell. Something similar holds for allocation of new adjustable reference cells: it is useful for the new internal value to depend on the internal value of some old reference cell. Thus, we also assume the following operation:

```
Parameter aref_new :
  ∀ R1 T1 (f1: R1 → T1) (r: aref f1)
    R2 T2 (f2 : R2 → T2) (g: ∀v, aref_get r = f1 v → R2)
    (PF: ∀x pfx y pfy, f2 (g x pfx) = f2 (g y pfy)),
    aref f2.
Axiom aref_new_val :
  ∀ R1 T1 (f1: R1 → T1) v R2 T2 (f2: R2 → T2) g PF,
    aref_new (aref_val f1 v) g PF
    = aref_val f2 (g v (aref_get_val f1 v)).
```

Having the internal value of a new reference cell depend on that of one old reference cell suffices for our examples, but it may easily be extended to making the internal value of the new cell depend on the internal values of a list of existing reference cells.

2.1 Logical Consistency of Adjustable References

In order to show that the new axioms about adjustable references are logically consistent, we present a very naive implementation satisfying them. We model the adjustable reference type as its internal representation type, and have `aref_get` apply the function f to it. Adjusting reads simply ignore the adjustment and behave as normal reads.

```
Definition aref R T (f : R → T) := R.
Definition aref_val R T (f: R → T) (v : R) := v.
Definition aref_get R T (f: R → T) (r : aref f) := f r.
Definition aref_getu R A B (f : R → A → B) (upd : R → A → R * B)
      (PF: ∀x a, f (fst (upd x a)) = f x)
      (PF': ∀x a, snd (upd x a) = f x a) := aref_get f.
Definition aref_new R1 T1 (f1: R1 → T1) (r: aref f1)
   R2 T2 (f2 : R2 → T2) (g: ∀v, aref_get f1 r = f1 v → R2)
   (PF: ∀x pfx y pfy, f2 (g x pfx) = f2 (g y pfy)) :=
   g r eq_refl.
```

With this representation, it is straighforward to prove the associated axioms: our proof scripts are 'one-liners.'

2.2 Extraction to Efficient Imperative Code

Coq's extraction mechanism [7] generates OCaml code from the Coq definitions by erasing proofs and inserting suitable type casts to work around OCaml's type system. For types and operations that are specified as parameters, such as the `aref`, Coq allows us to specify their OCaml implementations.

There is also a way, using the `Extraction Implicit` directive, to remove further arguments (besides the propositional ones), if we know that a certain argument will not be used by the OCaml implementation. We use this feature to remove the unused observation function argument `f` from `aref_val` and `aref_getu`, as well as `f1` and `f2` from `aref_new`.

In OCaml, we implement adjustable references as a mutable reference cells storing the representation type. A normal read, `aref_get`, just reads the contents of the cell and applies the observation function, f, to it. An adjusting read, `aref_getu`, applies the update function, u, instead, and updates the reference cell as appropriate. Finally, `aref_new` simply creates a new reference cell as expected. Unfortunately, extraction cannot fully remove the second argument of `g` despite it being a proposition; so we call `g` with a dummy second argument. Below, we show our OCaml implementation with the OCaml types in comments.

```
type ('r,'t) aref = 'r ref
let aref_val x = ref x      (* 'r → ('r,'t) ref *)
let aref_get f r = f !r     (* ('r → 't) → ('r,'t) aref → 't *)
let aref_getu u r a =
   let (v, b) = u !r a in r := v; b
   (* ('r → 'a → 'r * 'b) → ('r, 'a → 'b) aref → 'a → 'b *)
let aref_new r g = ref (g !r ())
   (* ('r1,'t1) aref → ('r1 → unit → 'r2) → ('r2,'t2) aref *)
```

3 Memoization Using Adjustable References

A simple use of adjustable references is in the memoization optimisation. Given a function $f : A \to B$, we construct the function `memo` f which is extensionally equal to f, but which caches the results of previous f invocations, so that if `memo` f is called with the same argument again, the cached version is used. To implement the cache, we also require a decidable equality on A, as well as a hash function mapping elements of A to machine integers. We use the Coq **Section** mechanism to avoid repeating these assumptions for every definition.

```
Section Memo.
Variables (A B : Type) (f : A → B).
Variable eqA : ∀x y : A, { x = y } + { x ≠ y }.
Variable hash : A → int.
```

The cache is just an array of pairs (a, b) such that $f(a) = b$. In Coq,

```
Definition cache :=
  { c : Parray.t (option (A * B)) |
      ∀ x a b, Parray.get c x = Some (a, b) → b = f a }.
```

where we assume a module, `Parray`, implementing functional arrays.

The main program, `memo`, creates an adjustable reference cell holding an initially empty cache that represents the function f, and then returns a function that does an adjusting read from that reference cell. The main work of the adjusting read is performed by the `memo_upd` function, which reads the cache to determine if it contains an appropriate entry (a, b): if so, it returns b; if not, it calculates $f(a)$ and stores $(a, f(a))$ into the cache. In case of a hash collision, for simplicity, we simply overwrite the old colliding array entry.

We define these operations using the **Program** feature of Coq which allows us to write functions in a natural style and emits missing proof obligations as goals to be proved interactively at the end of the definition. In this example, we are basically asked to show that the initial and updated arrays are valid caches. For simplicity, we omit these easy proofs.

```
Program Definition memo_upd (c : cache) (a : A) : (cache * B) :=
  let h :=  hash a in
  match Parray.get c h with
    | None ⇒
        let b := f a in (Parray.set c h (Some (a, b)), b)
    | Some (a’, b) ⇒
        if eqA a a’ then (c, b) else
        let b := f a in (Parray.set c h (Some (a, b)), b)
  end. ⟨...⟩

Program Definition memo :=
  let r := aref_val (fun c : cache ⇒ f)
             (Parray.create (Int.repr 100) None) in
  aref_getu memo_upd _ _ r. ⟨...⟩
```

We can now easily prove that the memo function is equivalent to f: we unfold the definition of `memo` and rewrite using two adjustable reference axioms.

```
Lemma memo_eq : memo = f.
Proof. by unfold memo; rewrite aref_getuE, aref_get_val. Qed.
```

Finally, we close the Coq section, which will parametrize all the functions and lemmas declared within the section by the variables `A`, `B`, `f`, `eqA`, and `hash`.

```
End Memo.
```

4 Union-Find Path Compression

As a second example, we implement the path compression optimisation, which is crucial for achieving good performance in the union-find algorithm [12].

The union-find data structure describes a partition of a finite set, and supports two operations: (1) `find` returning the representative of an element (such that two elements are in the same partition iff they have the same representative), and (2) `union` that coalesces two partitions.

The data-structure is organised as an upward pointing forest so that elements of the same partition belong to the same tree. In this setting, `find` follows the parent-pointing edges from its argument until it reaches the root of its tree, which it returns as the representative, whereas `union` simply adds an edge from the root of the one partition to the root of the other.

To achieve practically constant (inverse Ackerman) time per operation, `find` 'compresses' the paths during look up. There are many ways of doing so, the simplest being to make all the nodes along the path from the input node to the root point directly to the root.

Below, we implement two functions that do the path lookup: `get_aux` (simply returning the root) and `find_aux` (also returning the updated path-compressed graph). When writing these functions in Coq, we have (as an orthogonal problem) to prove termination for the path lookups. For conciseness, however, we omit these proofs from the presentation.

```
Definition get_rel (a : Parray.t int) (x y : int) : Prop :=
  x ≠ y ∧ Parray.get a y = x.

Program Fixpoint get_aux a x (WF : Acc (get_rel a) x) :=
  let y := Parray.get a x in
  if y ≡ x then x else get_aux a y ⟨...⟩.

Program Fixpoint find_aux a x (WF : Acc (get_rel a) x) :=
  let y := Parray.get a x in
  if y ≡ x then (a, x)
  else let '(f, r) := find_aux a y ⟨...⟩ in
       (Parray.set f x r, r).
```

Our main data structure then consists of a rank array returning an approximate size of each partition (so that when `union` merges two partitions, it makes the smaller point to the larger one) and an adjustable reference to the parent-pointing array representing the union-find forest. Formally, we represent the latter as a refinement type, as we need to ensure that it has the same length as the rank array and actually represents a forest (so that path lookups terminate).

```
Definition closed_arr_cond (length: int) (a: Parray.t int) :=
  Parray.length a = length
  ∧ (∀ x, Int.ltu (Parray.get a x) (Parray.length a))
  ∧ well_founded (get_rel a).

Definition closed_arr length := { a | closed_arr_cond length a }.

Record t := { ranks : Parray.t int ;
              parr : aref (@get_closed (Parray.length ranks)) }.
```

With these definitions, it is now easy to implement the main union-find operations. The omitted proof obligations have to do with ensuring 'forestness' is maintained when compressing paths or adding edges, and the soundness of `find_aux` with respect to `get_aux`.

```
Program Definition create (size: int) : t :=
  {| ranks := Parray.create size Int.zero ;
     parr := aref_val _ (Parray.init size id) |}.

Definition find (uf: t) : int → int :=
  aref_getu (fun a x ⇒ @find_aux (proj1_sig a) x ⟨...⟩) ⟨...⟩⟨...⟩
     (parr uf).

Definition union (uf : t) (a b : int) : t :=
  let a' := find uf a in
  let b' := find uf b in
  if a' ≡ b' then uf
  else
    let ra := Parray.get (ranks uf) a' in
    let rb := Parray.get (ranks uf) b' in
    if Int.ltu ra rb then
      {| ranks := Parray.set (ranks uf) b' (Int.add ra Int.one) ;
         parr := aref_new (parr uf) (fun r PF ⇒
            existT _ (Parray.set (proj1_sig r) b' a') ⟨...⟩) ⟨...⟩|}
    else
      {| ranks := Parray.set (ranks uf) a' (Int.add rb Int.one) ;
         parr := aref_new (parr uf) (fun r PF ⇒
            existT _ (Parray.set (proj1_sig r) a' b') ⟨...⟩) ⟨...⟩|}.
```

In our Coq development, we proceed further to prove various properties about `create`, `find`, and `union` which together entail the correctness of our union-find implementation. We also use this union-find data structure to implement an efficient certified separation logic satisfiability checker.

5 Conclusion

This paper has presented adjustable references, a referentially transparent data type that enables imperative programming in a local and semantically unobservable fashion. We have seen how adjustable references can be used to implement and verify memoization and path compression, two important optimisations that cannot be programmed in a purely functional style.

One clear omission from this paper is a formal proof that the efficient imperative OCaml implementation of §2.2 is equivalent to the naive one of §2.1. Formally, we have to prove contextual equivalence in the context of a language with higher-order state, and a type and effect system including abstract, recursive and dependent types. To show that the two implementations are equivalent, one would have to extend the proof techniques for showing contextual equivalence, such as Kripke logical relations [10] or relation transition systems [6], to this setting. This task is by no means trivial, and is left as future work.

This work was largely inspired by a paper by Conchon and Filliâtre [5], who built persistent array and union-find implementations, whose performance is very close to that of the standard imperative implementations. In that paper, Conchon and Filliâtre used Coq to verify a monadic encoding of a slightly simplified form of those implementations against an axiomatisation of ML references. Adjustable references allows us to make a step further and program the exact path-compressing union-find algorithm directly in Coq.

It should be noted, however, that adjustable references are not a panacea. For example, they cannot directly be used to program the Conchon and Filliâtre persistent array [5]. The issue is that their implementation performs a sequence of updates that temporarily change the externally observable values of reference cells only to restore them at the end of the sequence. By definition, adjustable references do not permit such value-changing updates. To program such persistent data structures we need a more general primitive that can take a sequence of updates to multiple references and check that the entire sequence does not alter the observable value of any individual cell.

In essence, what we would like to have is an adjustable state monad, within which unrestricted read-write access to the internal values of adjustable references is allowed, together a primitive operation for converting internally stateful computations into pure computations:

$$\texttt{runST} : \forall c : \mathsf{AdjStateMonad}\ A.\ c \text{ is logically pure} \to A\,.$$

The somewhat informal condition that c is logically pure is supposed to check that (1) c's output is independent of the internal values of any reference cells it accessed, and (2) c's end-to-end behaviour does not change the external values of any reference cells that were not newly created by its execution. Again, properly defining `runST` and formally justifying its soundness seems quite a challenging task, which we leave for future work.

Even though adjustable references as presented in this paper are clearly not applicable to every internally imperative, persistent data structure, we have identified and presented two examples that can easily be programmed with them. We hope that they will be equally useful in programming other similar persistent data structures directly inside interactive theorem provers.

Acknowledgements. I would like to thank Beta Ziliani and the anonymous ITP 2013 reviewers for their constructive feedback, which helped improve the presentation of the paper.

References

1. Appel, A. W., and Blazy, S.: Separation logic for small-step Cminor. In Schneider, K., and Brandt, J.: TPHOLs 2007. LNCS, vol. 4732, pp. 5-21. Springer, Heidelberg (2007)
2. Aydemir, B. E., Bohannon, A., Fairbairn, M., Foster, J. N., Pierce, B. C., Sewell, P., Vytiniotis, D., Washburn, G., Weirich, S., and Zdancewic, S.: Mechanized Metatheory for the Masses: The PoplMark Challenge. In Hurd, J., and Melham, T. F.: TPHOLs 2005. LNCS, vol. 3603, pp. 50-65. Springer, Heidelberg (2006)
3. Bertot, Y.: A short presentation of Coq. In Mohamed, O. A., Muñoz, C., and Tahar, S.: TPHOLs 2008. LNCS, vol. 5170, pp. 12-16. Springer, Heidelberg (2008)
4. Chlipala, A.: Mostly-automated verification of low-level programs in computational separation logic. In PLDI 2011, pp. 234-245. ACM (2011)
5. Conchon, S., and Filliâtre, J.-C.: A persistent union-find data structure. In Russo, C. V., Dreyer, D.: ML 2007, pp. 37-46. ACM (2007).
6. Hur, C., Dreyer, D., Neis, G., and Vafeiadis, V.: The marriage of bisimulations and Kripke logical relations. In POPL 2012, pp. 59-72. ACM (2012)
7. Letouzey, P.: A new extraction for Coq. In Geuvers, H., and Wiedijk, F.: TYPES 2002. LNCS, vol. 2646, pp. 200-219. Springer, Heidelberg (2002)
8. Nanevski, A., Morrisett, G., and Birkedal, L.: Hoare type theory, polymorphism and separation. J. Functional Programming 18(5-6): 865-911. CUP (2008)
9. Nanevski, A., Vafeiadis, V., and Berdine, J.: Structuring the verification of heap-manipulating programs. In POPL 2010, pp. 261-274. ACM (2010)
10. Pitts, A. M., and Stark, I. D. B.: Operational Reasoning for Functions with Local State. In Gordon, A. D., and Pitts, A. M. (eds): Higher Order Operational Techniques in Semantics, pp. 227-273. CUP (1998)
11. Slind, K., and Norrish, M.: A brief overview of HOL4. In Mohamed, O. A., Muñoz, C., and Tahar, S.: TPHOLs 2008. LNCS, vol. 5170, pp. 28-32. Springer, Heidelberg (2008)
12. Tarjan, R. E., and Van Leeuwen, J.: Worst-case analysis of set union algorithms. JACM 31(2):245-281. ACM (1984)
13. Wenzel, M., Paulson, L. C., and Nipkow, T.: The Isabelle Framework. In Mohamed, O. A., Muñoz, C., and Tahar, S.: TPHOLs 2008. LNCS, vol. 5170, pp. 33-38. Springer, Heidelberg (2008)
14. Yu, D., and Shao, Z.: Verification of safety properties for concurrent assembly code. In ICFP 2004, pp. 175-188. ACM (2004)