Program Extraction for Mutable Arrays

Kazuhiko Sakaguchi(✉)

University of Tsukuba, Tsukuba, Japan
sakaguchi@coins.tsukuba.ac.jp

Abstract. We present a mutable array programming library for the Coq proof assistant which enables simple reasoning method based on Ssreflect/Mathematical Components, and extractions of the efficient OCaml programs using in-place updates. To refine the performance of extracted programs, we improved the extraction plugin of Coq. The improvements are based on trivial transformations for purely functional programs and reduce the construction and destruction costs of (co)inductive objects, and function call costs effectively. As a concrete application for our library and the improved extraction plugin, we provide efficient implementations, proofs, and benchmarks of two algorithms: the union–find data structure and the quicksort algorithm.

1 Introduction

The program extraction mechanism [7–9,15] of the Coq proof assistant [20] is a code generation method for obtaining certified functional programs from constructive formal proofs and definitions by eliminating the non-informative part and widely used for developments of high-reliability software. For example, the CompCert project [6] uses Coq and its program extraction mechanism for obtaining a formally verified and executable C compiler which guarantees the correctness of the translation.

Verification and code generation (including extraction) of programs with side effects are important issues to apply proof assistants to realistic software developments. Particularly, in-place updates of mutable objects are important to increase the efficiency of programs. There are many recent studies to support writing, reasoning about, and generating programs with side effects in proof assistants, e.g., Coq (Ynot) [13], Isabelle/HOL [2], Idris [1], and F⋆ [17]. Ynot is such a representative Coq library which can handle various kind of side-effects, e.g., accessing reference cells, non-termination, throwing/catching exceptions, and I/O. Ynot is based on an axiomatic extension for Coq called Hoare Type Theory (HTT) [12] and supports reasoning with separation logic [14] which are good at its expressiveness, but not good at reducing proof burden in Coq.

This study establishes a novel, lightweight, and axiom-free method for verification and extraction of efficient programs using mutable arrays. Our library supports a simple monadic DSL for mutable array programming and powerful and simple reasoning method, and which is achieved by focusing only on mutable

J. P. Gallagher and M. Sulzmann (Eds.): FLOPS 2018, LNCS 10818, pp. 51–67, 2018.
https://doi.org/10.1007/978-3-319-90686-7_4

arrays and doing away with more side-effects such as reference cells and local states. Our contribution consists of three parts:

- In Sect. 3, we define a state monad specialized for mutable array programming—the array state monad—and give two interpretations of it for reasoning and program extraction. The former interpretation is defined in a purely functional way with building blocks taken from the Ssreflect/Mathematical Components (MathComp) library [11,21] and makes it possible to reduce the reasoning tasks on effectful programs to those on purely functional programs. The latter interpretation enables extraction of efficient effectful programs and provides encapsulation function like `runST` of state threads [5] which corresponds to the interpretation function in the former interpretation. The encapsulation mechanism converts effectful functions written by the array state monad to referential transparent functions and it also enables encapsulation of proofs.
- In Sect. 4, we present two new optimization techniques for the program extraction plugin. The optimizations are based on well-known transformations of purely functional programs, but effectively reduce the execution time of programs extracted with our library. More generally, the optimizations are especially effective for two cases: 1. proofs using mathematical structures and its theories provided by the MathComp library and 2. programs using monads that have functional types, e.g., State, Reader, and Continuation monads.
- In Sect. 5, we demonstrate elegant formalization techniques for programs using mutable arrays, and efficiency of the extracted programs using our library and the improved extraction, through the two applications—the union–find data structure and the quicksort algorithm.
  Our formalization of the quicksort algorithm uses the theory of permutations provided by the `perm` library of MathComp, and we show that some properties of permutations are also helpful for the reasoning of the quicksort algorithm. We also show benchmark results of the applications, and it indicates that performances of extracted programs using our library and the improved extraction are comparable to handwritten OCaml implementations of same algorithms and much better than purely functional implementations of the same kind of algorithms.

The source code of our library, the improved extraction plugin, case studies, benchmark scripts and patches for existing libraries are available at:

https://github.com/pi8027/efficient-finfun.

2 Finite Types and Finite Functions in Coq

This section briefly introduces the fintype and finfun libraries from the MathComp library with some modifications. The former provides an interface for types with finitely many elements—*finite types*. The latter provides a type of functions with finite domains—*finite functions* or *finfuns*—which is used as a

representation of arrays in this paper and relies on the former part. Key definitions and lemmas of the both libraries are listed in Table 1 and described below. In our previous work [16], we modified these libraries to improve the efficiency of code extracted from proofs.

2.1 A Finite Type Library—fintype

A finite type is a type with finitely many elements. The fintype library provides definitions of a class of finite types (`finType`) and its basic operations. The class of finite types is defined as a canonical structure [10] which contains a type and a witness of its finiteness. In the original fintype library, such finiteness of a type `T` is characterized by a duplicate-free enumeration of elements of `T`. In the modified one, it is recharacterized by a pair of the natural cardinal number `c` and a bijection between `T` and a finite ordinal type `'I_c = {0, ..., c - 1}`. The bijection is given by a pair of an encoding function of type `T → 'I_c` and a decoding function of type `'I_c → T`.

The two most basic operations on finite types are the enumeration (`enum`) and the cardinality (`#|_|`) of a subset of a finite type. We also provide accessors for the cardinal number and the bijection used by the new characterization of finiteness. The former is `$|_|` notation, and the latter is `fin_encode` and `fin_decode` functions. For any `T : finType`, `$|T|` is equal to `#|T|` and more efficiently computable than `#|T|`.

Many canonical `finType` instances are given by the MathComp library: `unit`, `bool`, finite ordinals `'I_n`, `option`, `sum`, `prod`, finite functions with a finite codomain, finite subsets `{set T}`, symmetric groups `{perm T}`, and more.

2.2 A Finite Function Library—finfun

The finfun library provides definitions of a type of finfuns (`{ffun T → U}`) and its basic operations. Finfuns from `T` to `U` is defined as `#|T|` tuples of `U`. Tuples are size-fixed lists defined in the tuple library, and easily translated to arrays in extracted programs by the `Extract Inductive` command.

```
Inductive finfun_type (T : finType) (U : Type) := Finfun of #|T|.-tuple U.
```

The two most basic operations of finfuns are the application (`fun_of_fin`) and the construction from CiC functions (`finfun`). The application `fun_of_fin f i` is the `(fin_encode i)`-th element of the underlying tuple of `f`. The construction `finfun f` is the finfun extensionally equal to `f`, and the underlying tuple of it associates `f (fin_encode i)` for each `i`-th element. Both of them can be efficiently computable by using `Array.get` and `Array.init` functions respectively in extracted OCaml programs.

The application `fun_of_fin f i` can be expressed as `f i` because `fun_of_fin` is the coercion from finfuns to CiC functions. The finfun library also provides constructor notations `[ffun i ⇒ e]` and `[ffun ⇒ e]` that are equivalent to `finfun (fun i ⇒ e)` and `finfun (fun _ ⇒ e)` respectively.

Table 1. Key definitions and lemmas in the modified fintype and finfun libraries

Coq judgement	Informal semantics
`T : finType`	`T` is a type with finitely many elements
`T : finType` ⊢ `Finite.sort T : Type`	*coercion* from `finType` to `Type`
`T : Type` ⊢ `[finType of T] : finType`	*canonical* `finType` instance of `T`
`n : nat` ⊢ `'I_n : Type`	type of natural numbers $\{0, \ldots, n-1\}$ (finite ordinal type)
`n : nat, i : 'I_n` ⊢ `nat_of_ord i : nat`	*coercion* from finite ordinal type to `nat`
`T : finType, pT : predType T, p : pT` ⊢ `enum p : seq T`	enumeration of the subset `p` of `T`
`T : finType, pT : predType T, p : pT` ⊢ `#|p| : nat`	cardinality of the subset `p` of `T`
`T : finType` ⊢ `#|T| : nat`	cardinality of the finite type `T`
`T : finType, x : T` ⊢ `fin_encode x : 'I_#|T|` `T : finType, i : 'I_#|T|` ⊢ `fin_decode i : T`	} bijection between `T` and `'I_#|T|`
`T : finType, x : T` ⊢ `fin_encodeK x : fin_decode (fin_encode x) = x`	`fin_decode` is a left inverse of `fin_encode`
`T : finType, i : 'I_#|T|` ⊢ `fin_decodeK i : fin_encode (fin_decode i) = i`	`fin_encode` is a left inverse of `fin_decode`
`T : finType, U : Type` ⊢ `{ffun T → U} : Type`	finfuns from `T` to `U`
`T : finType, U : Type, f : {ffun T → U}, i : T` ⊢ `fun_of_fin f i : U`	image of `i` under `f` (*coercion* from finfuns to CiC functions)
`T : finType, U : Type, f : T → U` ⊢ `finfun f : {ffun T → U}`	finfun such that extensionally equal to the CiC function `f`
`T : finType, U : Type, f g : {ffun T → U}` ⊢ `ffunP f g : f =1 g ↔ f = g`	functional extensionality of finfuns[a]
`T : finType, U : Type, f : T → U, x : T` ⊢ `ffunE f x : fun_of_fin (finfun f) x = f x`	unfolding lemma for application of finfuns

[a] `=1` means an extensional equality with arity 1.

3 Representing Mutable Arrays in Coq

The state monad [22] is useful to represent computations with states in purely functional languages such as Haskell and Coq. *Actions* of the state monad have types of the form $S \to S \times A$ where S is a state type and A is a return type. They take the initial state as its arguments, and returns the result of the type A paired with the final state. To represent various computations with mutable arrays, we need an *array state monad* that hold two conditions: (C1) it can handle multi-dimensional and multiple mutable arrays, and (C2) it never needs copy operations on arrays.

The former part of (C1)—handling multi-dimensional arrays—is achieved naturally by representing arrays by finfuns, because the finfuns of type $\{\mathsf{ffun}\ I_1 \times \cdots \times I_n \to A\}$ correspond to the multi-dimensional arrays of A indexed by $I_1, \ldots, I_n$.

Monad transformers are well-known methods to compose monadic effects, and the state monad transformer seem to be suitable for solving the latter part of (C1)—handling multiple arrays. However, if we allow one to compose the array state monad and other monads, the condition (C2) does not hold. For example, the actions of the monad that is a composition of the state monad transformer and the list monad have a type $S \to \mathrm{list}(S \times A)$, and it needs to copy the state on each branch of computation. Therefore, the array state monad should be defined in a more refined way.

We solve the above problem by defining the array state monad as an inductive data type that has a restricted set of primitive operations shown below:

```
Definition Sign : Type := seq (finType * Type).

Implicit Types (I J K : finType) (sig : Sign).

Inductive AState : Sign → Type → Type :=
  | astate_ret_ : ∀{sig} {A : Type}, A → AState sig A
  | astate_bind_ :
      ∀{sig} {A B : Type}, AState sig A → (A → AState sig B) → AState sig B
  | astate_lift_ :
      ∀{I} {T : Type} {sig} {A : Type}, AState sig A → AState ((I, T) :: sig) A
  | astate_GET_ : ∀{I} {T : Type} {sig}, 'I_#|I| → AState ((I, T) :: sig) T
  | astate_SET_ :
      ∀{I} {T : Type} {sig}, 'I_#|I| → T → AState ((I, T) :: sig) unit.
```

`AState [:: (I1, T1); ...; (In, Tn)] A` is a type of array state monad actions with `I1, ..., In` indexed mutable arrays of `T1, ..., Tn` respectively and the return type `A`. The first argument of `AState` is called a *signature* and indicates types of mutable arrays. Let Σ be a metavariable of signatures, and Σ_i means i-th element of the signature Σ. We refer to the array corresponds to the Σ_i as the *i-th array (of the signature Σ)*.

Each constructor of `AState` corresponds to return, bind, lift, get and set operators. The lift operator can lift array state monad actions of a signature Σ to that with a signature $(I, T) :: \Sigma$, and lifted actions does not affect the first array. Get and set operators can only access the first array. The lift operator

is necessary to get and set element of an array after the second, and it is also useful for modular programming.

We also define aliases for all constructors of `AState` to avoid the construction and destruction costs of tuples in extracted OCaml programs[1], e.g.:

```
Definition astate_ret {sig A} a := @astate_ret_ sig A a.
```

Primitive get and set operators take an index of type `'I_#|I|`. Indices of type `I` are also applicable by using the encoding function.

```
Notation astate_get i := (astate_GET (fin_encode i)).
Notation astate_set i x := (astate_SET (fin_encode i) x).
```

Programs represented by `AState` values cannot run directly. We give an interpretation for the array state monad by a translation to the functions of type $S \to S \times A$, where S is a type of mutable arrays defined as follows:

```
Fixpoint states_AState sig : Type :=
  if sig is (Ix, T) :: sig' then states_AState sig' * {ffun Ix → T} else unit.
```

`states_AState` takes a signature $[(I_1, T_1); \ldots; (I_n, T_n)]$, and returns a Cartesian product of all types of arrays in the signature: $\mathsf{unit} \times \{\mathsf{ffun}\ I_n \to T_n\} \times \cdots \times \{\mathsf{ffun}\ I_1 \to T_1\}$. We choose this order of types to omit parenthesis, because the $\times$ and $(\cdot, \cdot)$ operators have left associativity in Coq.

The translation is defined as follows:

```
Definition ffun_set
           (I : finType) (T : Type) (i : I) (x : T) (f : {ffun I → T}) :=
  [ffun j ⇒ if j == i then x else f j].

Definition runt_AState sig (A : Type) : Type :=
  states_AState sig → states_AState sig * A.

Definition run_AState : ∀sig A, AState sig A → runt_AState sig A :=
  @AState_rect (fun sig A _ ⇒ runt_AState sig A)
  (* return *) (fun _ _ a s ⇒ (s, a))
  (* bind *)   (fun _ _ _ _ f _ g s ⇒ let (s', a) := f s in g a s')
  (* lift *)   (fun _ _ _ _ _ f '(s, a) ⇒ let (s', x) := f s in (s', a, x))
  (* get *)    (fun _ _ _ i s ⇒ (s, s.2 (fin_decode i)))
  (* set *)    (fun _ _ _ i x '(s, a) ⇒ (s, ffun_set (fin_decode i) x a, tt)).
```

`ffun_set` is a pure set function for finfuns. It takes an index $i : T$, a value $x : A$ and a finfun $f : \{\mathsf{ffun}\ T \to A\}$, and returns a new finfun f' which is equal to f except that the i-th element is changed to x. `run_AState` is a interpretation function for the array state monad which is inductively defined on `AState` values.

3.1 Program Extraction for the Array State Monad

This section provides a method to extract efficient stateful OCaml programs from Coq proofs which use the array state monad. In another point of view, we

[1] In OCaml programs, arguments of constructors are parenthesized and comma separated. If a constructor is replaced with some function by the `Extraction Inductive` command, arguments of the constructor are interpreted as tuples.

give an another interpretation for array state monad by the OCaml program extraction.

In stateful settings, state propagation can be achieved by in-place updates instead of state monad style propagation, and moreover it is not needed to return a new state in each action. We defined the array state monad as an inductive data type only because to restrict primitive operations and its case analysis is never used except for `run_AState`. Thus we interpret array state monad actions as OCaml functions which take states and return its result by the `Extract Inductive` command:

```
Definition runt_AState_ sig (A : Type) : Type := states_AState sig → A.

Extract Inductive AState ⇒ "runt_AState_"
  [(* return *) " (fun a s -> a)"
   (* bind *)   " (fun (f, g) s -> let r = f s in g r s)"
   (* lift *)   " (fun f s -> let (ss, _) = Obj.magic s in f ss)"
   (* get *)    " (fun i s -> let (_, s1) = Obj.magic s in s1.(i))"
   (* set *)    " (fun (i, x) s -> let (_, s1) = Obj.magic s in s1.(i) <- x)"]
  "(* It is not permitted to use AState_rect in extracted code. *)".
```

We also give same realizations of aliases for the constructors of `AState`:

```
Extract Inlined Constant astate_ret ⇒ "(fun a s -> a)".
Extract Inlined Constant astate_bind ⇒ "(fun f g s -> let r = f s in g r s)".
...
```

Finally, we provide an encapsulation function for the array state monad as a realization of the `run_AState` function:

```
Extract Constant run_AState ⇒
  "(fun sign f s ->
    let rec copy sign s =
      match sign with
        | [] -> Obj.magic ()
        | _ :: sign' -> let (s', a) = Obj.magic s in
                        Obj.magic (copy sign' s', Array.copy a) in
    let s' = copy sign s in
    let r = Obj.magic f s' in (s', r))".
```

The encapsulation is achieved by duplicating all the input (initial) arrays by `Array.copy` and using the copied arrays in the execution of effectful actions. As a result, the range of in-place updates is limited to the copied arrays and it never affects outside of the `run_AState`.

3.2 Small Example: Swap Two Elements

Let us show a small example — swap — of programming and verification with the array state monad. The action swap(i, j) takes i-th and j-th values of the first array by `astate_get`, and then set them reversely by `astate_set`.

```
Definition swap (I : finType) {A : Type} {sig : Sign} (i j : I) :
  AState ((I, A) :: sig) unit :=
  x ← astate_get i; y ← astate_get j; astate_set i y;; astate_set j x.
```

`x ← t1; t2` and `t1;; t2` are "do"-like notations which are equivalent to `astate_bind t1 (fun x ⇒ t2)` and `astate_bind t1 (fun _ ⇒ t2)` respectively. Correctness of the swap action is described by the following lemma.

```
Lemma run_swap
      (I : finType) (A : Type) (sig : Sign) (i j : I)
      (f : {ffun I → A}) (fs : states_AState sig) :
  run_AState (swap i j) (fs, f) = (fs, [ffun k ⇒ f (tperm i j k)], tt).
```

`tperm i j : {perm I}` is a permutation (bijection) on `I` which transposes `i` and `j`. `tperm i j k : I` is `j` if `k = i`, `i` if `k = i` and otherwise `k`. This formulation is useful for reasoning on a sequence of `swap` actions, e.g., sorting algorithms.

Operations of the array state monad can be erased from the goal by the simplification tactic `rewrite /=`. More generally, the case analysis and the simplification work as the erasure.

```
...
  ============================
  (fs,
  ffun_set (fin_decode (fin_encode j)) (f (fin_decode (fin_encode i)))
    (ffun_set (fin_decode (fin_encode i)) (f (fin_decode (fin_encode j))) f),
  tt) = (fs, [ffun k ⇒ f ((tperm i j) k)], tt)
```

The encoding/decoding functions can be erased by the `fin_encodeK` lemma. Both sides of equation have the form of `(fs, _, tt)`, thus we use the congruence rule.

```
congr (_, _, _); rewrite !fin_encodeK.
```

```
...
  ============================
  ffun_set j (f i) (ffun_set i (f j) f) = [ffun k ⇒ f ((tperm i j) k)]
```

To prove a equation of finfuns, we use the lemma of functional extensionality `ffunP`. Applications of finfuns can be unfolded by the `ffunE` lemma.

```
apply/ffunP ⇒ k; rewrite !ffunE /=.
```

```
...
  k : I
  ============================
  (if k == j then f i else if k == i then f j else f k) = f ((tperm i j) k)
```

The remaining goal can be proved by case analysis on comparison and `tperm`.

```
case: tpermP; do!case: eqP; congruence. Qed.
```

The lift operator is also erased by similar method. Here is a correctness proof of the lifted swap action.

```
Global Opaque swap.
Lemma run_lift_swap
      (I I' : finType) (A B : Type) (sig : Sign) (i j : I)
      (f : {ffun I → A}) (g : {ffun I' → B}) (fs : states_AState sig) :
  run_AState (sig := [:: (I', B), (I, A) & sig])
             (astate_lift (swap i j)) (fs, f, g) =
  (fs, [ffun k ⇒ f (tperm i j k)], g, tt).
Proof. by rewrite /= run_swap. Qed.
```

4 Optimizations by an Improved Extraction Plugin

We provide two modifications for the extraction plugin to improve the efficiency of extracted programs, particularly which use the array state monad. The Coq program extraction translates Gallina[2] programs to target languages (OCaml, Haskell, Scheme, and JSON) and consists of three translations: 1. extraction from Gallina to MiniML, an intermediate abstract language for program extraction, 2. simplification (optimization) of MiniML terms, and 3. translation from MiniML to target languages.

The modifications are of the part 2 and 1 of the translation. The former reduces the construction and destruction costs of (co)inductive objects by inlining. The latter reduces the function call costs by applying η-expansion to match expressions and distributing the added arguments to each branch. Each optimization is particularly effective for programs using the MathComp library and monadic programs respectively.

4.1 Destructing Large Records

The MathComp library provides many mathematical structures [4] as canonical structures, e.g., `eqType`, `choiceType`, `countType`, `finType`, etc. which are represented as nested records in extracted programs. For example, the modified `finType` definition is translated to a nested record with 5 constructors and 11 fields by the program extraction. We implemented additional simplification mechanisms for MiniML terms to prevent performance degradation caused by handling such large records.

This section describes simplification rules for MiniML terms provided by the original and improved extraction plugin. The rules act on type coercion (`Obj.magic` in OCaml, and `unsafeCoerce` in Haskell) are omitted here because of simplicity. The key simplification rule for unfolding pattern matchings provided by the original extraction plugin is generalized ι-reduction relation.

Definition 1 (Generalized ι-reduction). *We inductively define an auxiliary relation $\rightsquigarrow_\iota^{\overline{cl}}$ for a sequence of pattern-matching clauses $\overline{cl} = C_1(\overline{x_1}) \to u_1 \mid \cdots \mid C_n(\overline{x_n}) \to u_n$ by the following three rules:*

$$C_i(t_1, \ldots, t_m) \rightsquigarrow_\iota^{\overline{cl}} \begin{array}{l}\mathsf{let}\ x_{i,1} := t_1\ \mathsf{in} \ldots \\ \mathsf{let}\ x_{i,m} := t_m\ \mathsf{in}\ u_i\end{array} \tag{1}$$

$$t \rightsquigarrow_\iota^{\overline{cl}} t' \Rightarrow \mathsf{let}\ x := u\ \mathsf{in}\ t \rightsquigarrow_\iota^{\overline{cl}} \mathsf{let}\ x := u\ \mathsf{in}\ t' \tag{2}$$

$$t_1 \rightsquigarrow_\iota^{\overline{cl}} t_1' \wedge \cdots \wedge t_m \rightsquigarrow_\iota^{\overline{cl}} t_m' \Rightarrow \begin{array}{l}\mathsf{match}\ t\ \mathsf{with} \\ \mid D_1(\overline{x_1}) \to t_1 \\ \mid \ldots \\ \mid D_m(\overline{x_m}) \to t_m\end{array} \rightsquigarrow_\iota^{\overline{cl}} \begin{array}{l}\mathsf{match}\ t\ \mathsf{with} \\ \mid D_1(\overline{x_1}) \to t_1' \\ \mid \ldots \\ \mid D_m(\overline{x_m}) \to t_m'\end{array} \tag{3}$$

The generalized ι-reduction relation $\rightsquigarrow_\iota$ is defined as follows:

$$t \rightsquigarrow_\iota^{\overline{cl}} t' \Rightarrow (\mathsf{match}\ t\ \mathsf{with}\ \overline{cl}) \rightsquigarrow_\iota t' \tag{4}$$

[2] The specification language of Coq.

The combination of the rules (1) and (4) provides simple ι-reduction. The rules (2) and (3) allows to traverse nested match and let expressions in the head of term t in the rule (4).

Other simplification rules are listed below.

$$(\lambda x.\, t)\, u \rightsquigarrow \mathsf{let}\ x := u\ \mathsf{in}\ t \tag{5}$$

$$\mathsf{let}\ x := u\ \mathsf{in}\ t \rightsquigarrow t[x := u] \quad \begin{array}{l}(t \text{ or } u \text{ is atomic ,} \\ \quad \text{or } x \text{ occurs at most once})\end{array} \tag{6}$$

$$(\mathsf{let}\ x := t_1\ \mathsf{in}\ t_2)\ u_1 \ldots u_n \rightsquigarrow \mathsf{let}\ x := t_1\ \mathsf{in}\ t_2\ u_1 \ldots u_n \tag{7}$$

$$\begin{array}{l}(\mathsf{match}\ t\ \mathsf{with} \\ \quad \mid C_1(\overline{x_1}) \to t_1 \\ \quad \mid \ldots \\ \quad \mid C_n(\overline{x_n}) \to t_n \\ )\, u_1 \ldots u_m\end{array} \rightsquigarrow \begin{array}{l}\mathsf{match}\ t\ \mathsf{with} \\ \quad \mid C_1(\overline{x_1}) \to t_1\, u_1 \ldots u_m \\ \quad \mid \ldots \\ \quad \mid C_n(\overline{x_n}) \to t_n\, u_1 \ldots u_m\end{array} \tag{8}$$

$$\begin{array}{l}\mathsf{match}\ t\ \mathsf{with} \\ \quad \mid C_1(\overline{x_1}) \to \lambda y_1 \ldots y_m.\, t_1 \\ \quad \mid \ldots \\ \quad \mid C_n(\overline{x_n}) \to \lambda y_1 \ldots y_m.\, t_n\end{array} \rightsquigarrow \begin{array}{l}\lambda y_1 \ldots y_m.\ \mathsf{match}\ t\ \mathsf{with} \\ \quad \mid C_1(\overline{x_1}) \to t_1 \\ \quad \mid \ldots \\ \quad \mid C_n(\overline{x_n}) \to t_n\end{array} \tag{9}$$

$$\mathsf{let}\ x := (\mathsf{let}\ y := t_1\ \mathsf{in}\ t_2)\ \mathsf{in}\ t_3 \rightsquigarrow \mathsf{let}\ y := t_1\ \mathsf{in}\ (\mathsf{let}\ x := t_2\ \mathsf{in}\ t_3) \tag{10}$$

$$\mathsf{let}\ x := C(t_1, \ldots, t_n)\ \mathsf{in}\ u \rightsquigarrow \begin{array}{l}\mathsf{let}\ y_1 := t_1\ \mathsf{in} \ldots \mathsf{let}\ y_n := t_n\ \mathsf{in} \\ \mathsf{let}\ x := C(y_1, \ldots, y_n)\ \mathsf{in}\ u'\end{array} \tag{11}$$

where u' in the rule (11) is obtained by replacing all the subterms of the form $(\mathsf{match}\ x\ \mathsf{with} \cdots \mid C(z_1, \ldots, z_n) \to t \mid \ldots)$ in the u with the term $t[z_1 := y_1, \ldots, z_n := y_n]$ which is a ι-reduced term of $(\mathsf{match}\ C(y_1, \ldots, y_n)\ \mathsf{with} \cdots \mid C(z_1, \ldots, z_n) \to t \mid \ldots)$.

All simplification rules are safe for purely functional programs, but not safe for OCaml programs generally. The rules (6), (8) and (9) may change the execution order of code and skip executing some code, and other rules also help to apply these rules. Therefore it is difficult to implement these rules as optimizations for OCaml programs, and it is appropriate to implement it as a MiniML optimizer.

The generalized ι-reduction without the rule (2) and the rules (5) through (9) are provided by the original extraction plugin. We additionally implemented the rules (2), (10), and (11) to it. The rule (11) recursively destructs nested records, and the rule (10) assist it in the case of a term of the form $(\mathsf{let}\ x := (\mathsf{let}\ y := t\ \mathsf{in}\ C(t_1, \ldots, t_n))\ \mathsf{in}\ \ldots)$ occurred.

Let us exemplify a simplification process of the following definition[3]:

```
Definition example : nat → nat → nat → bool :=
  let T := nat_eqType in fun x y z : T ⇒ (x == y) || (x == z) || (y == z).
```

[3] `nat_eqType` is the canonical `eqType` instance of `nat`.

The simplification process is as follows. Constants inlined by extraction are underlined here. Identifiers `nat_eqType`, `Equality.Pack`, and `Equality.Mixin` are abbreviated here as $\texttt{nat}_{\texttt{eq}}$, $\texttt{Pack}_{\texttt{eq}}$, and $\texttt{Mixin}_{\texttt{eq}}$.

$$
\begin{aligned}
& \mathsf{let}\ T := \underline{\texttt{nat}_{\texttt{eq}}}\ \mathsf{in} \\
& \lambda x\, y\, z.\, \underline{\texttt{eq\_op}}\ T\ x\ y \vee \underline{\texttt{eq\_op}}\ T\ x\ z \vee \underline{\texttt{eq\_op}}\ T\ y\ z \\
= {} & \mathsf{let}\ T := \texttt{Pack}_{\texttt{eq}}\ (\texttt{Mixin}_{\texttt{eq}}\ \texttt{eqn}\ \ldots)\ \mathsf{in}\ \lambda x\, y\, z. && \text{(unfold)} \\
& \quad (\lambda T.\, \mathsf{match}\ T\ \mathsf{with}\ \texttt{Pack}_{\texttt{eq}}\ (\texttt{Mixin}_{\texttt{eq}}\ f\ \_) \to f)\ T\ x\ y \\
& \vee (\lambda T.\, \mathsf{match}\ T\ \mathsf{with}\ \texttt{Pack}_{\texttt{eq}}\ (\texttt{Mixin}_{\texttt{eq}}\ f\ \_) \to f)\ T\ x\ z \\
& \vee (\lambda T.\, \mathsf{match}\ T\ \mathsf{with}\ \texttt{Pack}_{\texttt{eq}}\ (\texttt{Mixin}_{\texttt{eq}}\ f\ \_) \to f)\ T\ y\ z \\
\rightsquigarrow^* {} & \mathsf{let}\ T := \texttt{Pack}_{\texttt{eq}}\ (\texttt{Mixin}_{\texttt{eq}}\ \texttt{eqn}\ \ldots)\ \mathsf{in} && \text{(rules (5)} \\
& \lambda x\, y\, z.\, (\mathsf{match}\ T\ \mathsf{with}\ \texttt{Pack}_{\texttt{eq}}\ (\texttt{Mixin}_{\texttt{eq}}\ f\ \_) \to f)\ x\ y && \text{and (6))} \\
& \quad \vee\ (\mathsf{match}\ T\ \mathsf{with}\ \texttt{Pack}_{\texttt{eq}}\ (\texttt{Mixin}_{\texttt{eq}}\ f\ \_) \to f)\ x\ z \\
& \quad \vee\ (\mathsf{match}\ T\ \mathsf{with}\ \texttt{Pack}_{\texttt{eq}}\ (\texttt{Mixin}_{\texttt{eq}}\ f\ \_) \to f)\ y\ z \\
\rightsquigarrow {} & \mathsf{let}\ a := \texttt{Mixin}_{\texttt{eq}}\ \texttt{eqn}\ \ldots\ \mathsf{in}\ \mathsf{let}\ T := \texttt{Pack}_{\texttt{eq}}\ a\ \mathsf{in} && \text{(rule (11))} \\
& \lambda x\, y\, z.\, (\mathsf{match}\ a\ \mathsf{with}\ \texttt{Mixin}_{\texttt{eq}}\ f\ \_ \to f)\ x\ y \\
& \quad \vee\ (\mathsf{match}\ a\ \mathsf{with}\ \texttt{Mixin}_{\texttt{eq}}\ f\ \_ \to f)\ x\ z \\
& \quad \vee\ (\mathsf{match}\ a\ \mathsf{with}\ \texttt{Mixin}_{\texttt{eq}}\ f\ \_ \to f)\ y\ z \\
\rightsquigarrow {} & \mathsf{let}\ c := \texttt{eqn}\ \mathsf{in}\ \mathsf{let}\ b := \ldots\ \mathsf{in}\ \mathsf{let}\ a := \texttt{Mixin}_{\texttt{eq}}\ c\ b\ \mathsf{in} && \text{(rule (11))} \\
& \mathsf{let}\ T := \texttt{Pack}_{\texttt{eq}}\ a\ \mathsf{in}\ \lambda x\, y\, z.\, c\ x\ y \vee c\ x\ z \vee c\ y\ z \\
\rightsquigarrow^* {} & \lambda x\, y\, z.\, \texttt{eqn}\ x\ y \vee \texttt{eqn}\ x\ z \vee \texttt{eqn}\ y\ z && \text{(rule (6))}
\end{aligned}
$$

`T` can be unfolded by rules (6) and (4) if `T` has occurred only once. However, we need the rule (11) to apply the ι-reduction over the let expressions which cannot be simplified by the rule (6).

4.2 η-expansion on Case Analysis

Match expressions returning a function are commonly used in monadic programming and dependently typed programming. However, they increase the number of function calls and closure allocations and decrease the performance of programs. Let us consider the following monadic program that branches depending on whether the integer state is even or odd:

$$\mathsf{get} \gg\!= \lambda n : \mathbb{Z}.\, \mathsf{if}\ n \bmod 2 = 0\ \mathsf{then}\ f\ \mathsf{else}\ g.$$

By unfolding the get and $\gg\!=$ in the above program, a match expression returning a function[4] can be found:

$$\lambda n : \mathbb{Z}.\, (\mathsf{if}\ n \bmod 2 = 0\ \mathsf{then}\ f\ \mathsf{else}\ g)\, n.$$

Another example from dependently typed programming is a map function for size-fixed vectors:

[4] `if` expressions are syntax sugar for match expressions in Coq.

```
Fixpoint vec (n : nat) (A : Type) : Type :=
  if n is S n' then (A * vec n' A) else unit.

Fixpoint vmap (A B : Type) (f : A → B) (n : nat) : vec n A → vec n B :=
  if n is S n' then fun '(h, t) ⇒ (f h, vmap f t) else fun _ ⇒ tt.
```

The match expressions in the above examples can be optimized by the rules (8) and (9) respectively, and those rules can be applied for many other cases. However, these rules are not complete because of its syntactic restriction: match expressions to which that rules are applied should have following arguments or have λ-abstractions for each branch. Therefore, we apply the full η-expansion on all match expressions in the process of extraction from Gallina to MiniML.[5] The rule (8) can be applied for η-expanded match expressions.

We additionally provide new Vernacular command `Extract Type Arity` to declare the arity of an inductive type which is realized by the `Extract Inductive` command, because the Coq system cannot recognize that the arity of `AState` is 1. Declared arities are used for full η-expansion described above. This command can be used as follows: `Extract Type Arity AState 1.`

5 Case Studies

This section demonstrates our library and improved extraction plugin using two applications: the union–find data structure and the quicksort algorithm. Here we provide an overview of formalizations and the performance comparison between the extracted code and other implementations for each application.

All the benchmark programs were compiled by OCaml 4.05.0+flambda with optimization flags `-O3 -remove-unused-arguments -unbox-closures` and performed on a Intel Core i5-7260U CPU @ 2.20 GHz equipped with 32 GB of RAM. Full major garbage collection and heap compaction are invoked before each measurement. All benchmark results represent the median of 5 different measurements with same parameters.

5.1 The Union–find Data Structure

We implemented and verified the union–find data structure with the path compression and the weighted union rule [18,19]. The formalization takes 586 lines, and most key properties and reasoning on the union–find can be easily written out by using the path and fingraph library.

The benchmark results are shown in the Fig. 1. Here we compare the execution times of the OCaml code extracted from above formalization (optimized and unoptimized[6] version) and handwritten OCaml implementation of same algorithm. The procedure to be measured here is the sequence of union n times

[5] Implementing it as a part of the simplification of MiniML terms is difficult, because MiniML is a type-free language.

[6] It is extracted by disabling new optimization mechanisms described in Sect. 4, but compiled with same OCaml compiler and optimization flags.

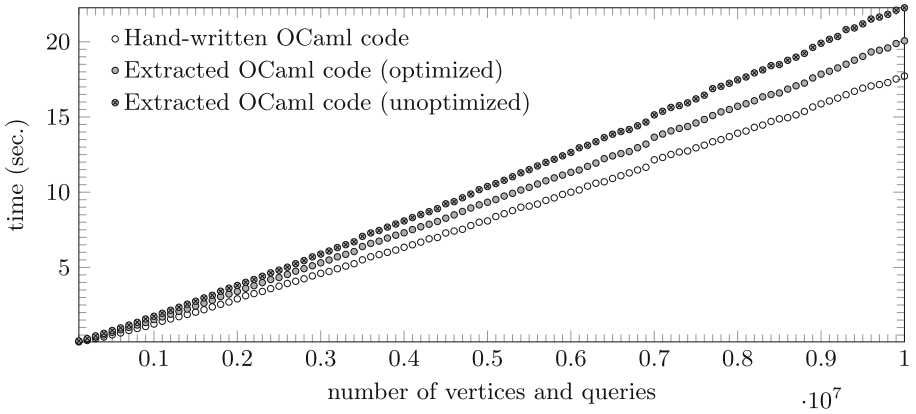

Fig. 1. Benchmark results of the union–find data structure

and find n times on n vertices union–find data structure, where all parameters of unions and finds are randomly selected. The time complexity of this procedure is $\Theta(n\alpha(n,n))$ where α is a functional inverse of Ackermann's function. The results indicate that the OCaml implementation is about 1.1 times faster than the optimized Coq implementation, the optimized Coq implementation is about 1.1 times faster than the unoptimized Coq implementation, and the execution times of all implementations increase slightly more than linear.

5.2 The Quicksort Algorithm

We implemented and verified the quicksort algorithm by using the array state monad. The formalization including partitioning and the upward/downward search takes 365 lines, and the key properties are elegantly written out by using the theory of permutations. Here we explain the formalization techniques used for the quicksort algorithm by taking the partitioning function as an example. The partitioning function for `I` indexed arrays of `A` and comparison function `cmp : A → A → bool`[7] has the following type:

```
partition : A → ∀i j : 'I_#|I|.+1, i ≤ j → AState [:: (I, A)] 'I_#|I|.+1
```

The `partition` takes a pivot of type and range of partition represented by indices `i j : 'I_#|I|.+1`, reorders the elements of the arrays from index `i` to `j - 1` so that elements less than the pivot come before all the other elements, and returns the partition position. We proved the correctness of `partition` as the following lemma:

[7] Here we assume that `cmp` is a total order and means "less than or equal to" in some sense.

```
CoInductive partition_spec
            (pivot : A) (i j : 'I_#|I|.+1) (arr : {ffun I → A}) :
  unit * {ffun I → A} * 'I_#|I|.+1 → Prop :=
  PartitionSpec (p : {perm I}) (k : 'I_#|I|.+1) :
  let arr' := [ffun ix ⇒ arr (p ix)] in
  (* 1 *) i ≤ k ≤ j →
  (* 2 *) perm_on [set ix | i ≤ fin_encode ix < j] p →
  (* 3 *) (∀ix : 'I_#|I|, i ≤ ix < k → ¬ cmp pivot (arr' (fin_decode ix))) →
  (* 4 *) (∀ix : 'I_#|I|, k ≤ ix < j → cmp pivot (arr' (fin_decode ix))) →
          partition_spec pivot i j arr (tt, arr', k).

Lemma run_partition
      (pivot : A) (i j : 'I_#|I|.+1) (Hij : i ≤ j) (arr : {ffun I → A}) :
  partition_spec pivot i j arr (run_AState (partition pivot Hij) (tt, arr)).
```

The `run_partition` can be applied for goals including some executions of `partition` without giving concrete parameters by using the simple idiom `case: run_partition`, and it obtains the properties of `partition` described below. It is achieved by a Ssreflect convention [11, Sect. 4.2.1], which means separating the specification from the lemma as the coinductive type family `partition_spec`.

In the specification `partition_spec`, the finfun `arr`, permutation `p`, and index `k` indicates the initial state, permutation performed by the partition, and partition position respectively, and the final state `arr'` is represented by

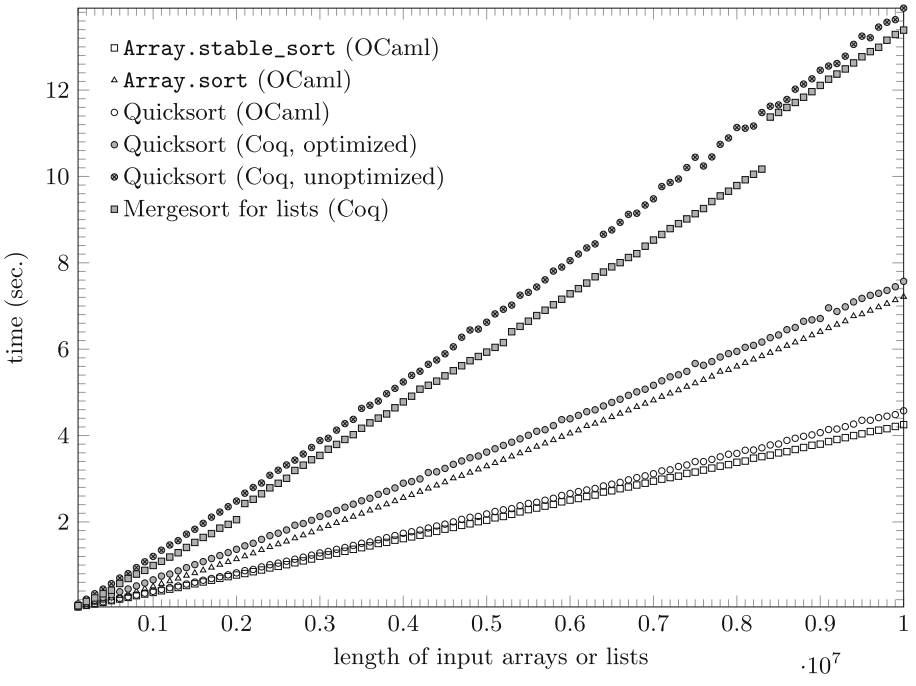

Fig. 2. Benchmark results of the quicksort and mergesort

`[ffun ix ⇒ arr (p ix)]` which means the finfun `arr` permuted by `p`. Properties of the partition are given as arguments of `PartitionSpec` and numbered from `1` to `4` in above code. Each property has the following meaning:

1. the partition position `k` is between `i` and `j`,
2. the partition only replaces value in the range from `i` to `j - 1`,
3. values in the range from `i` to `k - 1` are less than the pivot, and
4. values in the range from `k` to `j - 1` are greater than or equal to the pivot.

Of particular interest is that the property 2 can be written in the form of `perm_on A p` which means that the permutation `p` only displaces elements of `A`. `perm_on` is used for constructing algebraic theory in the MathComp library, but is also helpful for reasoning on sorting algorithms.

The benchmark results are shown in the Fig. 2. Here we compare the execution times of following implementations of sorting algorithms for randomly generated arrays or lists of integers: 1. `Array.stable_sort` and 2. `Array.sort` taken from OCaml standard library, 3. handwritten OCaml implementation of the quicksort, 4. optimized and 5. unoptimized OCaml code extracted from above formalization, and 6. OCaml code extracted from a Coq implementation of the bottom-up merge-sort algorithm for lists. The results indicate that the implementation 5 (quicksort in Coq, unoptimized) is slowest of those, the implementation 4 (quicksort in Coq, optimized) is 1.7–1.8 times faster than implementations 5 and 6, and OCaml implementations 1, 2 and 3 are 1.07–1.8 times faster than the implementation 3.

6 Related Work

Ynot [13] is a representative Coq library for verification and extraction of higher-order imperative programs. Ynot supports various kind of side-effects, separation-logic-based reasoning method, and automation facility [3] for it, and provides many example implementations of and formal proofs for data structures and algorithms including the union–find data structure. The formal development of the union–find provided by Ynot takes 1,067 lines of code, and our formal development of it (see Sect. 5.1) takes 586 lines. Both implementations use almost the same optimization strategies: the union by rank and the weighted union rule respectively, and the path compression. This comparison indicates that Ynot is good at its expressiveness, but our method has smaller proof burden.

7 Conclusion

We have established a novel, lightweight, and axiom-free method for verification and extraction of efficient effectful programs using mutable arrays in Coq. This method consists of the following two parts: a state monad specialized for mutable array programming (the array state monad) and an improved extraction plugin for Coq. The former enables a simple reasoning method for and safe extraction of efficient effectful programs. The latter optimizes programs extracted from formal developments using our library, and it is also effective for mathematical structures provided by the MathComp library and monadic programs.

We would like to improve this study with more expressive array state monad, large and realistic examples, and correctness proof of our extraction method for effectful programs.

Acknowledgments. We thank Yukiyoshi Kameyama and anonymous referees for valuable comments on an earlier version of this paper. This work was supported by JSPS KAKENHI Grant Number 17J01683.

References

1. Brady, E.: Programming and reasoning with algebraic effects and dependent types. In: ICFP 2013, pp. 133–144. ACM (2013)
2. Bulwahn, L., Krauss, A., Haftmann, F., Erkök, L., Matthews, J.: Imperative functional programming with Isabelle/HOL. In: Mohamed, O.A., Muñoz, C., Tahar, S. (eds.) TPHOLs 2008. LNCS, vol. 5170, pp. 134–149. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-71067-7_14
3. Chlipala, A., Malecha, G., Morrisett, G., Shinnar, A., Wisnesky, R.: Effective interactive proofs for higher-order imperative programs. In: ICFP 2009, pp. 79–90. ACM (2009)
4. Garillot, F., Gonthier, G., Mahboubi, A., Rideau, L.: Packaging mathematical structures. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 327–342. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03359-9_23
5. Launchbury, J., Peyton Jones, S.L.: Lazy functional state threads. In: PLDI 1994, pp. 24–35. ACM (1994)
6. Leroy, X.: A formally verified compiler back-end. J. Autom. Reason. **43**(4), 363–446 (2009)
7. Letouzey, P.: A new extraction for Coq. In: Geuvers, H., Wiedijk, F. (eds.) TYPES 2002. LNCS, vol. 2646, pp. 200–219. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-39185-1_12
8. Letouzey, P.: Programmation fonctionnelle certifiée - L'extraction de programmes dans l'assistant Coq. Ph.D. thesis, Université Paris-Sud (2004)
9. Letouzey, P.: Extraction in Coq: an overview. In: Beckmann, A., Dimitracopoulos, C., Löwe, B. (eds.) CiE 2008. LNCS, vol. 5028, pp. 359–369. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-69407-6_39
10. Mahboubi, A., Tassi, E.: Canonical structures for the working Coq user. In: Blazy, S., Paulin-Mohring, C., Pichardie, D. (eds.) ITP 2013. LNCS, vol. 7998, pp. 19–34. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39634-2_5
11. Mahboubi, A., Tassi, E.: Mathematical components (2016). https://math-comp.github.io/mcb/book.pdf
12. Nanevski, A., Morrisett, G., Birkedal, L.: Hoare type theory, polymorphism and separation. J. Funct. Prog **18**(5–6), 865–911 (2008)
13. Nanevski, A., Morrisett, G., Shinnar, A., Govereau, P., Birkedal, L.: Ynot: dependent types for imperative programs. In: ICFP 2008, pp. 229–240. ACM (2008)
14. O'Hearn, P., Reynolds, J., Yang, H.: Local reasoning about programs that alter data structures. In: Fribourg, L. (ed.) CSL 2001. LNCS, vol. 2142, pp. 1–19. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-44802-0_1
15. Paulin-Mohring, C.: Extracting F_ω's programs from proofs in the Calculus of Constructions. In: POPL 1989, pp. 89–104. ACM (1989)

16. Sakaguchi, K., Kameyama, Y.: Efficient finite-domain function library for the Coq proof assistant. IPSJ Trans. Prog. **10**(1), 14–28 (2017)
17. Swamy, N., Hriţcu, C., Keller, C., Rastogi, A., Delignat-Lavaud, A., Forest, S., Bhargavan, K., Fournet, C., Strub, P.Y., Kohlweiss, M., Zinzindohoue, J.K., Zanella-Béguelin, S.: Dependent types and multi-monadic effects in F⋆. In: POPL 2016, pp. 256–270. ACM (2016)
18. Tarjan, R.E.: Efficiency of a good but not linear set union algorithm. J. ACM **22**(2), 215–225 (1975)
19. Tarjan, R.E., van Leeuwen, J.: Worst-case analysis of set union algorithms. J. ACM **31**(2), 245–281 (1984)
20. The Coq Development Team: The Coq Proof Assistant Reference Manual (2017). https://coq.inria.fr/distrib/V8.7.0/refman/
21. The Mathematical Components Project: The mathematical components repository. https://github.com/math-comp/math-comp
22. Wadler, P.: Monads for functional programming. In: Jeuring, J., Meijer, E. (eds.) AFP 1995. LNCS, vol. 925, pp. 24–52. Springer, Heidelberg (1995). https://doi.org/10.1007/3-540-59451-5_2