# Program Extraction for Mutable Arrays

Kazuhiko Sakaguchi

*University of Tsukuba, Tennodai 1-1-1, Tsukuba, Japan*

**Abstract**

We present a lightweight method to represent, verify, and extract efficient programs involving mutable arrays in the Coq proof assistant. Our method mainly consists of a library for handling mutable arrays and an improved extraction plugin. Our library provides a monadic domain specific language for modeling computations involving mutable arrays, a simple reasoning method based on the Ssreflect extension and the Mathematical Components library, and an extraction method to efficient OCaml programs using in-place updates. Our extraction plugin improves the performance of our extracted programs, or more appropriately, through the application of simple program transformations for purely functional programs, it reduces both construction and destruction costs of inductive and coinductive objects and function call costs. As concrete applications for our method, we provide efficient implementations, correctness proofs, and benchmarks of the union–find data structure and the quicksort algorithm.

*Keywords:* interactive theorem proving, formally verified software, Coq, program extraction, program transformation and optimization, mutable arrays, state monad

## 1. Introduction

The program extraction mechanism [1, 2, 3, 4] of the Coq proof assistant [5] is a code generation method for obtaining certified functional programs from constructive formal proofs by eliminating the non-informative parts. Code generation methods of proof assistants including program extraction are effective for developing highly-reliable software systems. For example, the CompCert project [6] uses Coq and its program extraction mechanism to obtain a formally verified optimizing C compiler.

Verification and code generation of programs using side effects are important issues when applying proof assistants to realistic software developments. In particular, in-place updates of mutable objects are important to increase the efficiency of programs.

This study establishes a lightweight method to represent, verify, and extract efficient programs involving mutable arrays. Our library supports a simple monadic domain specific language (DSL) for mutable array programming and a powerful but simple reasoning method which is achieved by focusing only on mutable arrays and doing away with other side-effects such as reference cells and local states. This article explores the following topics:

*Email address:* sakaguchi@logic.cs.tsukuba.ac.jp (Kazuhiko Sakaguchi)

- In Section 2, we introduce the fintype, tuple, and finfun sub-libraries of the Mathematical Components (MathComp) library [7, 8] and the modifications that we propose. These sub-libraries are originally made for providing fundamental definitions and properties for finite sets and finite functions, and are mainly used for representing and manipulating several kinds of mathematical objects (e.g., finite subsets, matrices, and finite groups), but can also be used as a good abstraction to represent and reason about arrays.

  Our modifications to these libraries are originally devised in our previous work [9] to increase the performance of random access of immutable arrays in extracted programs. In this study, we have updated this modification strategy itself to obtain better compatibility with the original MathComp library. In Section 2.4, we explain this new modification strategy, which is generally applicable and useful to extend an existing mathematical structure with additional operators and axioms without changing its expressiveness.

- In Section 3, we define a state monad to model computations involving mutable arrays and give its two interpretations for reasoning and program extraction. The former one is defined in a purely functional way with the building blocks from Section 2 and makes it possible to turn the verification of effectful programs into the verification of purely functional programs. The latter enables us to extract efficient and effectful programs involving mutable arrays. The program extraction interpretation also encapsulates the side effects as in the `runST` function of the ST (state thread) monad [10]. The encapsulation mechanism converts effectful functions written in the state monad into referentially transparent functions. This is achieved by giving an original realization for the interpretation (run) function.

- In Section 5, we improve the optimization facility of the extraction plugin with a few additional techniques. Most of these optimizations are based on trivial transformations rules of purely functional programs but drastically increase the efficiency of programs extracted with our library. The optimizations are especially effective in two cases:

  1 proofs using mathematical structures and its theories provided by the MathComp library and

  2 programs using monads that have functional types, such as the State, Reader, and Continuation monads.

- In Section 6, two applications, the union–find data structure and the quicksort algorithm, are used to show how programs using mutable arrays can be formalized in our new environment and how efficient their extracted programs are.

  Our formalization of the quicksort algorithm uses the theory of permutations provided by the `perm` sub-library of MathComp, and we give the key properties of permutations for reasoning about the quicksort algorithm. We also present some benchmark results that indicate that our extracted programs run much faster than purely functional implementations and are comparable to handwritten OCaml implementations.

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.

A conference version of this article appeared in the proceedings of the 14th International Symposium of Functional and Logic Programming [11]. The present version has been improved with a new modification method for the MathComp library which achieves better compatibility with the original one (Section 2) and a relaxed definition of our monadic DSL for mutable array programming (Section 3.2). A new encapsulation mechanism is required as a consequence of this relaxation (Section 3.4). We obtain simpler and more efficient extracted programs (Section 6) and achieve more modularity than the previous version.

Coq is an interactive system. Its command language is called Vernacular. In this article, all Coq outputs (e.g., current goals, extracted code, and types of given terms) produced by Vernacular commands are marked with a light gray background.

## 2. Finite types and finite functions in Coq

This section introduces the fintype, tuple, and finfun sub-libraries of the MathComp library and describes some modifications that we propose. The fintype library provides an interface for types with finitely many elements—*finite types*. The finfun library provides a theory of functions with finite domains—*finite functions* or *finfuns*. The tuple library provides a theory of fixed-size lists (tuples) which is used to define finite functions. This article uses finite functions and finite types as the representations of arrays and their index sets respectively. Key definitions and lemmas of those libraries are listed in Table 1 and described below.

Our modification is originally presented in our previous work [9] to improve the efficiency of extracted programs where finite functions are seen as immutable arrays. We have slightly updated this modification method in this article to get better compatibility with the original MathComp library. As a result, the formal proofs of the four-color theorem [12] and the odd order theorem [13] can be checked with the modified MathComp library without any change. Thus, most existing formal definitions and proofs which use the MathComp library could be checked with the modified MathComp library and benefit from our most efficient program extraction.

### *2.1. A finite type library—fintype*

A finite type is a type with finitely many elements. The fintype library provides a definition of a class of finite types (`finType`) and its basic operations. The class of finite types is defined by using the packed class design pattern [14][8, Section 7.3] and a canonical structure [15] which represents a type together with a witness of its finiteness. The original definition of `finType` is as follows.[1]

[1] The actual definition of `finType` is more complicated because the implementation of its inheritances requires a lot of coercions and canonical projections and relies on some workarounds to mitigate efficiency issues on type inference/checking.

Table 1: Key definitions and lemmas in the fintype and finfun sub-libraries in the modified MathComp library

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

[a] The definitions and lemmas we introduced by our modification are highlighted by $\star$.
[b] `T : finType, U : Type, n : nat` is used as assumptions in common for every line except the first line.
[c] `=1` means an extensional equality with arity 1. This is not an axiom and can be proved constructively.

```
Module Finite.

Definition axiom (T : eqType) e := ∀x : T, count_mem x e = 1.

Record mixin_of (T : eqType) := Mixin {
  mixin_enum : seq T;
  _ : axiom mixin_enum
}.

Record class_of (T : Type) := Class {
  base : Countable.class_of T;
  mixin : mixin_of (Equality.Pack base T)
}.

Structure type : Type := Pack { sort : Type; _ : class_of sort }.

End Finite.
```

The `mixin_of T` record contains operations and their properties that `finType` introduces: a list `mixin_enum` and a proof of type `axiom mixin_enum` stating that `mixin_enum` is a duplicate-free enumeration of elements of `T`. The `class_of` record bundles all the operations and their properties of `finType` and its superclasses (`countType`, `choiceType`, and `eqType`). The `Finite.type` structure[2] represents a type equipped with `finType` operators and axioms, and has a synonym `finType`.

The original `fintype` library provides two basic operations on finite types: the enumeration (`enum`) and the cardinality (`#|_|`) of a subset of a finite type. The subsets of a finite type `T` used for those operations are represented by `predType T` that is the generic interface for types equipped with a membership operation. The enumeration and the cardinality can be computed by filtering `mixin_enum` with the membership operation and counting its elements respectively.

In our modified version, we have extended `finType` with a bijection from `T : finType` to a finite ordinal type `ordinal n` or equivalently `'I_n` which consists of elements `{0, ..., n - 1}` and is defined as follows.

```
Inductive ordinal (n : nat) : predArgType :=
  Ordinal : ∀m : nat, m < n → ordinal n.
```

where the type `predArgType` is convertible with `Type` and is used to declare coercions from types.

The extension for `finType` is given by the following new `mixin_of` record.

```
Record mixin_of (T : eqType) := Mixin {
  (*1*) mixin_enum : seq T;
  (*2*) mixin_card : nat;
  (*3*) mixin_enc : T → 'I_mixin_card;
  (*4*) mixin_dec : 'I_mixin_card → T;
  (*5*) _ : ∀x, x ∈ mixin_enum;
  (*6*) _ : ord_enum mixin_card = map mixin_enc mixin_enum;
  (*7*) _ : mixin_enum = map mixin_dec (ord_enum mixin_card);
}.
```

[2]The `Structure` Vernacular command is just a synonym of the `Record` command and used to indicate that some canonical projections involving the record it defines will be declared later.

There are three new operations on `T : finType` other than the enumeration in the new `mixin_of` record:

(2) the cardinal number of `T`,

(3) an encoding function from `T` to `'I_mixin_card`, and

(4) a decoding function from `'I_mixin_card` to `T`.

The bijection is given as a function itself (3) and its inverse (4). Their properties states that

(5) `mixin_enum` contains all the elements of `T`,

(6) `mixin_enc` returns `i : 'I_mixin_card` for a given argument `x : T` where `x` is the `i`-th element of `mixin_enum`, and

(7) `mixin_dec` returns the `i`-th element of `mixin_enum` for a given argument `i : 'I_mixin_card`.

Both the original and the modified `mixin_of` record characterize the same notion of finiteness and do not state any other properties. In other words, one could derive an instance of either the original or the modified `mixin_of` record from an instance of the other one. Thus, this modification does not lose any instance of the `finType` structure, nor definitions and lemmas about `finType`. However, this modification provides canonical indexing functions which associate elements of finite types with finite ordinals one-to-one. For a carefully defined finite type instance, that indexing function can be reasonably fast and typically have a constant-time or logarithmic-time complexity.

The modified `fintype` library provides accessors for `mixin_card`, `mixin_enc`, and `mixin_dec` in the modified `mixin_of` record as the `$|_|` notation, `fin_encode`, and `fin_decode` respectively. For any `T : finType`, `$|T|` is equal to `#|T|` and more efficiently computable than `#|T|` but not applicable for subsets of finite types. The accessor functions for the encoding/decoding functions are defined for both types of ordinals (`'I_#|T|` or `'I_$|T|`). The latter ones are named as `Finite.encode` and `Finite.decode`.

The MathComp library provides many canonical `finType` instances including `unit`, `bool`, finite ordinals `'I_n`, `option`, `sum`, `prod`, finite functions with a finite codomain, finite subsets `{set T}`, and symmetric groups `{perm T}`. In our modified version, have defined most of these standard `finType`s with efficiently computable encoding and decoding functions.

### *2.2. A fixed-size list library—tuple*

Fixed-size lists (tuples) we introduce now are equivalent to vectors, a typical introductory example of dependent typed data structure. They are not defined as an inductive type carrying size information but rather as a record containing an underlying list and a size refinement.

```
Structure tuple_of (n : nat) (T : Type) : Type :=
  Tuple { tval :> seq T; _ : size tval == n }.
```

The type `tuple_of n T` (or equivalently `n.-tuple of T`) represents lists of elements of type `T` whose size is `n`.

We instruct the extractor to realize tuples by arrays with an `Extract Inductive` command. The type `n.-tuple of T` has only one informative field of type `seq T`, therefore we should put `Array.of_list` and `Array.to_list` [16, Section 26.2] as the OCaml realizations of constructor and destructor of tuples.

```
Extract Inductive tuple_of ⇒
  "array" ["Array.of_list"] "(fun f t -> f (Array.to_list t))".
```

Using `Array.of_list` and `Array.to_list` is a very inefficient way to construct and access elements of tuples. Fortunately, basic operations of tuples `tnth`, `mktuple`, and `codom_tuple` used in the finfun library can be done without using lists in extracted programs. For a tuple `t : n.-tuple T` and an ordinal `i : 'I_n`, `tnth t i` is the `i`-th element of `t`. For a function `f : 'I_n → T'`, `mktuple f` is a tuple of size `n` whose `i`-th element is `f i`. For a finite type `T` and a function `f : T → T'`, `codom_tuple f` is a tuple of size `#|T|` whose underlying list is `map f (enum T)` or `i`-th element is `f (fin_decode i)`.

```
tnth : ∀(n : nat) (T : Type), n.-tuple T → 'I_n → T
mktuple : ∀(n : nat) (T' : Type), ('I_n → T') → n.-tuple T'
codom_tuple : ∀(T : finType) (T' : Type), (T → T') → #|T|.-tuple T'
```

The two functions `tnth` and `mktuple` can be realized by `Array.get` and `Array.init` respectively and efficiently computable in extracted OCaml programs.

```
Extract Constant tnth ⇒ "Array.get".
Extract Constant mktuple ⇒ "Array.init".
```

The last function `codom_tuple` can also be realized using `Array.init` and `fin_decode`.

### *2.3. A finite function library—finfun*

The finfun library provides definitions of a type of finite functions `finfun_type aT rT` or equivalently `{ffun aT → rT}` and its theory. Finite functions are defined as tuples as follows.

```
Record finfun_type (aT : finType) (rT : Type) : predArgType :=
  Finfun { fgraph : #|aT|.-tuple of rT }.
```

The original `finfun` library provides two basic operations on finite functions: the application (`fun_of_fin`) and the construction from plain Coq functions (`finfun`). They are defined as follows.

```
Definition fun_of_fin
           (aT : finType) (rT : Type) (f : {ffun aT → rT}) (x : aT) : rT :=
  tnth (fgraph f) (enum_rank x)

Definition finfun (aT : finType) (rT : Type) (f : aT → rT) : {ffun aT → rT} :=
  @Finfun aT rT (codom_tuple f)
```

The function `enum_rank` is a function from `finType` elements to ordinals. It is extentionally equal to `fin_encode` but inefficient because it computes an index from the enumeration list. The application `fun_of_fin f i` returns the `(fin_encode i)`-th element of the underlying tuple of `f`. The construction `finfun f` is the finite function

extensionally equal to `f`, and the underlying tuple of it associates `f (fin_encode i)` for each `i`-th element. We have redefined them to mitigate efficiency issues in the modified `finfun` library as follows.

```
Definition fun_of_fin
           (aT : finType) (rT : Type) (f : {ffun aT → rT}) (x : aT) : rT :=
  tnth (fgraph f) (fin_encode x).

Definition finfun (aT : finType) (rT : Type) (f : aT → rT) : {ffun aT → rT} :=
  @Finfun aT rT (tcast (raw_cardE aT) (mktuple (fun i ⇒ f (Finite.decode i)))).
```

The new definitions of `fun_of_fin` and `finfun` are obtained by replacing `enum_rank` and `codom_tuple` with `fin_encode` and an equivalent expression using `mktuple` respectively. A tuple `mktuple (fun i ⇒ f (Finite.decode i)) : $|aT|.-tuple of rT` has the same underlying list as `codom_tuple f : #|aT|.-tuple of rT`. To match those types, `finfun` uses a casting function `tcast` and a proof `raw_cardE aT : $|aT| = #|aT|`. By using `tcast` and `Finite.decode` rather than `fin_decode`, the first (implicit) argument of `mktuple` can be `$|aT|` that can be computed more efficiently than `#|aT|`.

The application `fun_of_fin f i` can be expressed as `f i` because `fun_of_fin` is the coercion from finite functions to plain Coq functions. The finfun library also provides constructor notations `[ffun i ⇒ e]` and `[ffun ⇒ e]` which mean `finfun (fun i ⇒ e)` and `finfun (fun _ ⇒ e)` respectively.

Now we are able to use finite functions as a good abstraction for (immutable) arrays. Elements of OCaml arrays of size $n$ are indexed by non-negative integers $0, \ldots, n-1$ but elements of finite functions of type `{ffun aT → rT}` are indexed by the finite type `aT`. Actual indices of the underlying array of a finite function are computed by the encoding function of the domain finite type `aT`. For example, sometimes we represent matrices and even higher-dimensional arrays as one-dimensional arrays, using simple arithmetic operations to compute indices. However, such arithmetic expressions can be implicitly derived from canonical instances of finite types by representing `aT_1, ..., aT_n` indexed arrays of `rT` as finite functions of type `{ffun aT_1 * ... * aT_n → rT}`. Additionally, boundary checking with regards to arrays is done by type checking; thus, random accesses do not exceed the array's boundaries.

### *2.4. Compatibility with the original library*

Modifying definitions without breaking existing proofs is a generally difficult task in formal verification. Fortunately, the modified MathComp library has achieved high-level compatibility with the original MathComp library. As a result, only 6 files in the MathComp library had to be changed, 5 of them (seq, fintype, tuple, finfun, and finset) were changed for performance reasons, and the remaining one (burnside_app) required a one-line patch on its proof. Moreover, the formal proofs of the four-color theorem [12] and the odd order theorem [13] can be checked without any change.

In our previous work [9], we removed the `mixin_enum` field from the `mixin_of` record to minimize the witness of finiteness because the duplicate-free enumeration could be obtained from the encoding/decoding functions. The accessor `Finite.enum` of `mixin_enum` was declared as an opaque constant by using the following module type `EnumSig` and the module `EnumDef`.[3]

[3] Using `<:` instead of `:` makes the module transparent.

```
(* In the ‘Finite‘ module. *)
Module Type EnumSig.
Parameter enum : ∀cT : type, seq cT.
Axiom enumDef : enum = fun cT ⇒ mixin_enum (class cT).
End EnumSig.

Module EnumDef : EnumSig.
Definition enum cT := mixin_enum (class cT).
Definition enumDef := erefl enum.
End EnumDef.

Notation enum := EnumDef.enum.

(* After the ‘Finite‘ module. *)
Canonical finEnum_unlock := Unlockable Finite.EnumDef.enumDef.
```

The body of this constant could then be accessed using the locking/unlocking facility of Ssreflect. The constants `fun_of_fin` and `finfun` were made opaque and locked in the same way. This abstraction mechanism made existing proofs very robust from changes of `finType` instances. However, the formal proof of the odd order theorem contained few unlockings and they needed to be patched.

Keeping both original and new characterizations in `mixin_of` as we have shown in Section 2.1 is a better practice to achieve high-level compatibility when modifying packed classes. This modification methodology duplicates constants and operators characterizing a mathematical structure (in our case, the enumeration and the encoding/decoding functions in `finType`). Fortunately, the packed class design pattern allows to (and sometimes should) contain such duplications. For example, `Finite.class_of` should (and does) contain a choice function and a counting function which can be constructed from an enumeration. Those duplications are necessary to makes `[choiceType of T]` and `[countType of T]` convertible with `[choiceType of [finType of T]]` and `[countType of [finType of T]]` respectively where `T` is a `Type` and has canonical instances of `choiceType`, `countType`, and `finType`. Generally, the `class_of` record should contain all the mixins of superclasses for this reason.

## 3. Mutable arrays in Coq

In this section, we present an approach based on the state monad [17] to represent, verify, and extract various computations involving mutable arrays in Coq.

### *3.1. State monad*

The state monad is a monad used to represent computations using an updatable state in functional programming and it is particularly useful in purely functional languages such as Haskell and Coq. This section reviews the notion of monads and the state monad.

A monad is a triple of a type constructor $M$ : Type $\to$ Type, a unary function $\mathit{return} : \alpha \to M\,\alpha$, and a binary function $(\gg\!=) : M\,\alpha \to (\alpha \to M\,\beta) \to M\,\beta$ called *bind*

which satisfies following axioms, called the *monad laws*:

$$\begin{aligned} \mathit{return}\, x \gg\!= f &= f\, x, \\ m \gg\!= \mathit{return} &= m, \\ (m \gg\!= f) \gg\!= g &= m \gg\!= (\lambda x.\, f\, x \gg\!= g). \end{aligned}$$

The terms of type $M\, A$ are called the monadic *actions* (of the monad $M$ and the return type $A$) in this article. Monadic actions can be sequentially composed as in procedural programming by using the bind function. Intuitively, $m \gg\!= f$ is a monadic action which first invokes $m$ and then applies $f$ to the return value of the first step.

The state monad is an instance of monad, parameterized by the type $S$ of the state to carry, and a triple $(\mathrm{State}, \mathit{return}, \gg\!=)$ consisting of the following:

$$\begin{aligned} \mathrm{State}\ A &:= S \to A \times S, \\ \mathit{return}\, a &:= \lambda s.\, (a, s), \\ m \gg\!= f &:= \lambda s.\, \mathsf{let}\ (a, s') := m\, s\ \mathsf{in}\ f\, a\, s'. \end{aligned}$$

The state monad actions take an initial state of the type $S$ as their argument and return a result of the type $A$ paired with a final state. It can easily be proved that the above triple satisfies the monad laws by expressing the laws with extensional equality instead of extensional equality or relying on the functional extensionality axiom.

The following *get* and *put* functions can be used to get and update the current state in the state monad.

$$\begin{aligned} &\mathit{get} : \mathrm{State}\ S && \mathit{put} : S \to \mathrm{State\ unit} \\ &\mathit{get} := \lambda s.\, (s, s) && \mathit{put}\, s := \lambda\, s'.\, (\mathrm{tt}, s) \end{aligned}$$

where the value tt is the only value of the unit type in Coq.

### *3.2. The array state monad*

The state monad defined in the previous section allows reading, writing, and duplicating the whole state freely. It is obviously difficult to give a safe syntactic translation from terms of the state monad to imperative programs involving a mutable state. We ideally want to allow only reading and writing the value of a given location of an array rather than the whole state to model only imperative-style computations involving mutable arrays. Therefore, we precisely redefine the state monad we need by restricting the possible operations with an inductive type definition as follows. Hereafter we call it the *array state monad*.

```
Inductive AState (I : finType) (T : Type) : Type → Type :=
  | astate_ret (A : Type) : A → AState I T A
  | astate_bind (A B : Type) :
    AState I T A → (A → AState I T B) → AState I T B
  | astate_get : I → AState I T T
  | astate_set : I → T → AState I T unit.
```

The type `AState I T A` is the type of monadic actions with return type `A` which may read and write an `I`-indexed array of `T`. The first two constructors are the monadic return and

bind operators respectively. The last two constructors are the random access operators for the array which can be only performed element-wise.

These monadic actions cannot be executed directly because they are defined by an inductive type definition. Therefore we give an interpretation of the monadic actions by a translation function from monadic actions of type `AState I T A` to functions of type `{ffun I → T} → A * {ffun I → T}` 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 run_AState I T :
  ∀A, AState I T A → {ffun I → T} → A * {ffun I → T} :=
  @AState_rect _ _ (fun A _ ⇒ {ffun I → T} → A * {ffun I → T})
  (* return *) (fun _ a s ⇒ (a, s))
  (* bind *)   (fun _ _ _ f _ g s ⇒ let (a, s') := f s in g a s')
  (* GET *)    (fun i s ⇒ (s i, s))
  (* SET *)    (fun i x s ⇒ (tt, ffun_set i x s)).
```

The function `ffun_set` is a purely functional implementation of the update function for finite functions. It takes an index $i : T$, a value $x : A$, and a finite function $f : \{\mathsf{ffun}\ T \to A\}$, and returns a new finite function $f'$ which is equal to $f$ except that the $i$-th element is changed to $x$.

The array state monad as defined above can handle only one array in which all the elements have the same type but more realistic computations may involve multiple mutable arrays which have different types of elements. Additionally each call of the above random access operators should use the encoding function of `I : finType` in extracted programs because they use `I` as the type of indices. Finite functions give us a good abstraction for arrays but encoding functions are sometimes computationally costly and needless. Therefore we extend and refine the `AState` type as follows.

```
Inductive AState : Type → Type → Type :=
  | astate_ret_ (S A : Type) : A → AState S A
  | astate_bind_ (S A B : Type) : AState S A → (A → AState S B) → AState S B
  | astate_frameL_ (Sl Sr A : Type) : AState Sl A → AState (Sl * Sr) A
  | astate_frameR_ (Sl Sr A : Type) : AState Sr A → AState (Sl * Sr) A
  | astate_GET_ (I : finType) (T : Type) : 'I_#|I| → AState {ffun I → T} T
  | astate_SET_ (I : finType) (T : Type) :
    'I_#|I| → T → AState {ffun I → T} unit.
```

The type `AState S A` is the type of array state monad which can handle multiple arrays with different types of elements. The type `S` of the state to carry can be an arbitrary type in the above definition but actually should be a nested Cartesian product of finite functions, or equivalently have the following shape.

$$\texttt{S} := \texttt{[ffun I → T]} \mid \texttt{S * S} \tag{1}$$

We introduce this constraint more formally in Section 3.4.

We have added two new constructors in the above definition: `astate_frameL_` and `astate_frameR_`. These constructors are used to access left-and right-hand sides of the state respectively when the state is a pair. We have also changed the type of the first arguments of the random access operators from `I` to `'I_#|I|`. It is still possible to

use indices of type `I` for random access through the encoding function. The following notations provide a convenient way to do this.

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

where the definitions `astate_GET` and `astate_SET` are the aliases for `astate_GET_` and `astate_SET_` respectively and will be defined in the next section.

The interpretation function can then be redefined as follows.

```
Definition run_AState_raw : ∀S A, AState S A → S → A * S :=
  @AState_rect (fun S A _ ⇒ S → A * S)
  (* return *) (fun _ _ a s ⇒ (a, s))
  (* bind *)   (fun _ _ _ _ f _ g s ⇒ let (a, s') := f s in g a s')
  (* frameL *) (fun _ _ _ _ f '(sl, sr) ⇒
                  let (a, sl') := f sl in (a, (sl', sr)))
  (* frameR *) (fun _ _ _ _ f '(sl, sr) ⇒
                  let (a, sr') := f sr in (a, (sl, sr')))
  (* GET *)    (fun _ _ i s ⇒ (s (fin_decode i), s))
  (* SET *)    (fun _ _ i x s ⇒ (tt, ffun_set (fin_decode i) x s)).
```

This array state monad is very similar to operational monads [18], whose motivation is to give the user the possibility to create new monads without caring about the monad laws. However, our motivation for using this style of definition is to provide an exact model of imperative-style computations involving mutable arrays and ensure the safety of extraction. Syntactically, the above definition of `AState` does not verify the monad laws in contrast to operational monads but, semantically, monad laws as properties can be proved by evaluating objects of type `AState` with `run_AState_raw`.

### *3.3. Program extraction for the array state monad*

This section provides a method to extract efficient OCaml programs destructively updating arrays from Coq proofs using the array state monad. In other words, we give another interpretation of the array state monad actions for efficient execution by the OCaml program extraction.

In stateful settings, state propagation can be achieved by in-place updates and reusing the same state instead of state monad style propagation. Moreover, it is not necessary to return a new state in each monadic action. We define the array state monad as an inductive type only to restrict its primitive operations and its case analysis is never used except for the definition of `run_AState`. Thus we translate the array state monad actions of type `AState S A` to OCaml functions of type `S -> A` as follows.

```
Extract Inductive AState ⇒ "(->)"
  [(* return *) "(fun a s -> a)"
   (* bind *)   "(fun (f, g) s -> let r = f s in g r s)"
   (* frameL *) "(fun f (sl, _) -> f sl)"
   (* frameR *) "(fun f (_, sr) -> f sr)"
   (* GET *)    "(fun i s -> s.(i))"
   (* SET *)    "(fun (i, x) s -> s.(i) <- x)"]
  "(* It is not permitted to use AState_rect in extracted programs. *)".
```

In OCaml, constructors are not functions and not curried, and more than one arguments of a constructor should be parenthesized and comma separated. If constructors are replaced with functions by use of a `Extraction Inductive` command, arguments of

the constructors are interpreted as tuples as we can see in the above realizations of `astate_bind` and `astate_SET`. Thus we also define aliases for all the constructors of `AState` and give them the same realizations as the constructors to avoid the construction and destruction costs of tuples in extracted OCaml programs.

```
Definition astate_bind {S A B} := @astate_bind_ S A B.
Definition astate_SET {I T} := @astate_SET_ I T.

Extract Inlined Constant astate_bind ⇒ "(fun f g s -> let r = f s in g r s)".
Extract Inlined Constant astate_SET ⇒ "(fun i x s -> s.(i) <- x)".
```

The first argument (`I : finType`) of random access operators is informative because `finType` is defined as a canonical structure having the sort `Type` and informative fields but are never used in extracted programs. Thus, we eliminate them by the following declarations.

```
Extraction Implicit astate_GET_ [I].
Extraction Implicit astate_SET_ [I].
Extraction Implicit astate_GET [I].
Extraction Implicit astate_SET [I].
```

We give the appropriate realization for `run_AState_raw` to match its return type with its Coq definition.

```
Extract Inlined Constant run_AState_raw ⇒ "(fun a s -> (a s, s))".
```

### *3.4. Encapsulation of effects*

Gallina is the specification language of Coq. Usually, Gallina terms and their extracted ML programs are pure and total. Therefore we should take care of referential transparency of the effectful ML program fragments extracted by our method. For example, the following definition `x` and its extracted program have different execution results because of side effects.

```
Definition x :=
  let a := [ffun _ : unit ⇒ 0] in
  (run_AState_raw
     (astate_bind (astate_set tt 1) (fun ⇒ astate_ret (a tt))) a).1.
```

The Coq definition of `x` can be proved to be equal to `0` but the extracted program returns `1`. The simplified extracted program of `x` is as follows.

```
let x =
  (*1*) let a = [|0|] in
  (*2*) fst ((a.(0) <- 1; a.(0)), a)
```

It initializes the array `a` as `[|0|]` at `(*1*)`, then executes the monadic action `(fun s -> s.(0) <- 1; a.(0))` with the initial array `a` at `(*2*)`. The action sets the `0`th (or equivalently `tt`-th) element of the array to `1` in the first step, then returns the `0`th element of `a` in the second step. In the Coq definition of `x`, the array read in the second step of the monadic action is `a` and is not the state carried in the array state monad. These arrays should be distinguished but both of them are `a` and not distinguished in the extracted program. This is what causes the problem. Therefore we need to copy the initial array before entering the execution of monadic actions. A correct version of the extracted program of `x` is then be

```
let x =
  let a = [|0|] in
  fst (let a' = Array.copy a in
       ((a'.(0) <- 1; a.(0)), a'))
```

Actually, we need more general copying function(s) for states whose types are defined by (1). More precisely, a type family corresponding to equation (1) can be defined by an inductive type definition and a fixpoint definition.

```
Inductive states_shape : Type :=
  | ffun_state of finType & Type
  | prod_state of states_shape & states_shape.

Fixpoint type_of_shape (shape : states_shape) : Type :=
  match shape with
    | ffun_state T U ⇒ {ffun T → U}
    | prod_state shape shape' ⇒ type_of_shape shape * type_of_shape shape'
  end.
```

The inductive type `states_shape` represents equation (1) and the recursive function `type_of_shape` computes a type from its value. A general copying function can then be defined

```
Definition ffun_copy
           (I : finType) (T : Type) (f : {ffun I → T}) : {ffun I → T} := f.

Fixpoint copy_state (shape : states_shape) :
  type_of_shape shape → type_of_shape shape :=
  match shape with
    | ffun_state T U ⇒ @ffun_copy T U
    | prod_state _ _ ⇒ fun '(s, s') ⇒ (copy_state s, copy_state s')
  end.
```

with the corresponding Vernacular commands for extraction.

```
Extraction Implicit ffun_copy [I].
Extract Inlined Constant ffun_copy ⇒ "Array.copy".
```

However, this extracted general copying function is unnecessarily complicated and inefficient because the extraction plugin cannot inline recursive functions. For this reason, we prefer to derive general copying functions by defining a class of types equipped with a copying function using the packed class design pattern [14, 15]. We first declare a mixin module `CopyableMixin` which has the following module type.

```
Module Type CopyableMixinSig.
Parameters
  (mixin_of : Type → Type)
  (copy : ∀(T : Type), mixin_of T → T → T)
  (copyE : ∀(T : Type) (C : mixin_of T) (x : T), copy C x = x)
  (ffun_mixin : ∀(I : finType) (T : Type), mixin_of {ffun I → T})
  (prod_mixin :
    ∀(T1 T2 : Type) (C1 : mixin_of T1) (C2 : mixin_of T2), mixin_of (T1 * T2)).
End CopyableMixinSig.
```

The `CopyableMixin` module is made opaque by declaring it with the command `Module CopyableMixin : CopyableMixinSig`: the fields of the `CopyableMixin` module cannot

be unfolded, and only the fields declared in the module type are visible outside the module. The mixin type `mixin_of` is defined as a record having the fields of the copying function and its correctness proof, but mixins for types other than the nested product type of finite functions are not definable, as a consequence of this opaqueness.

We then define the class of types named as `copyType` equipped with a copying function `copy : ∀T : copyType, T → T` by using the above mixin and the following canonical structure.

```
(* In the ‘Copyable‘ module. *)
Structure type : Type := Pack {sort; _ : CopyableMixin.mixin_of sort }.
```

We also provide an encapsulating run function `run_AState` as follows.

```
Definition run_AState
           (S : copyType) (A : Type) (m : AState S A) (s : S) : A * S :=
  run_AState_raw m (copy s).
```

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 in-place updates of `run_AState` are then only perfomed on copied input arrays. Of course, the function `run_AState_raw` must not be called directly to respect this encapsulation mechanism.

A very simple and efficient implementation of the encapsulating run function can be extracted with the use of some inlining directives. For example, the following definition

```
Definition run_3tuple (I : finType) (T A : Type)
                      (f g h : {ffun I → T}) (m : AState _ A) :=
  run_AState m (f, g, h).
```

is extracted to the following OCaml program.

```
Extraction run_3tuple.
```

```
(** val run_3tuple :
    Finite.coq_type -> 'a1 finfun_of -> 'a1 finfun_of -> 'a1 finfun_of ->
    (Copyable.sort->'a2) -> 'a2 * Copyable.sort **)

let run_3tuple _ f g h m =
  (fun a s -> (a s, s)) m
    (Obj.magic
      ((Obj.magic ((Obj.magic Array.copy f),(Obj.magic Array.copy g))),
      (Obj.magic Array.copy h)))
```

The type `Finite.coq_type` is the `finType` (`Finite.type`) structure in Coq. As `type` is a keyword in OCaml, the extractor renames it to `coq_type`. The function `Obj.magic` is the untyped identity function and disappears during compilation. The type `Copyable.sort` is a record projection of `Copyable.type` in Coq and extracted to the universal type `Obj.t`. Generally, those type coercions are inserted when overloaded functions using packed classes are applied to concrete instances. Section 8.3 provides detailed explanation of this issue.

## 4. Examples of programming, verification, and extraction with the array state monad

This section presents small examples of programming, verification, and extraction with the array state monad. It demonstrates that it is well suited for programming and doing verification. It also illustrates some limitations encountered with the original extraction plugin.

### *4.1. Swapping two elements*

Let us take a very small example:

```
Definition oswap (I : finType) {A : Type} (i j : 'I_#|I|) :
  AState {ffun I → A} unit :=
  mlet x := astate_GET i in
  mlet y := astate_GET j in
  astate_SET i y;; astate_SET j x.
```

The action `oswap i j` takes the $i$-th and $j$-th values of the array by `astate_GET`, and then sets them reversely by `astate_SET`. The monadic notations `mlet x := t1 in t2` and `t1 ;; t2` are equivalent to `astate_bind t1 (fun x ⇒ t2)` and `astate_bind t1 (fun _ ⇒ t2)` respectively. Correctness of the `oswap` action is described by the following lemma.

```
Definition perm_ffun
           (I : finType) (A : Type) (p : {perm I}) (f : {ffun I → A}) :=
  [ffun i ⇒ f (p i)].

Lemma run_oswap (I : finType) (A : Type) (i j : 'I_#|I|) (f : {ffun I → A}) :
  run_AState (oswap i j) f =
  (tt, perm_ffun (tperm (fin_decode i) (fin_decode j)) f).
```

The type `{perm T}` is the type of permutations on `T : finType` whose elements are coercible to functions (bijections) of type `T → T`. For any `T : finType` and `i j k : T`, a permutation `tperm i j : {perm I}` is the transposition of `i` and `j`; `tperm i j k` is `j` if `k = i`, `i` if `k = j`, and `k` otherwise. The function `perm_ffun p f` is the finite function `f` permuted by `p`. This formulation is useful to reason about algorithms that repetitively call `oswap` actions. It will be used when studying sorting algorithms (see Section 6.2).

When proving `run_oswap`, the initial goal is

```
  ...
   ============================
   run_AState (oswap i j) f =
   (tt, perm_ffun (tperm (fin_decode i) (fin_decode j)) f).
```

Operations on the array state monad can be simplified by rewriting them using the set of theorems `AStateE`. Executing

```
rewrite !run_AStateE.
```

returns the new goal

```
...
  ============================
  (tt,
  ffun_set (fin_decode j) (f (fin_decode i))
    (ffun_set (fin_decode i) (f (fin_decode j)) f)) =
  (tt, perm_ffun (tperm (fin_decode i) (fin_decode j)) f)
```

Both sides of equation now have the form of `(tt, _)`, thus we apply the congruence rule.

```
congr pair.
```

```
...
  ============================
  ffun_set (fin_decode j) (f (fin_decode i))
    (ffun_set (fin_decode i) (f (fin_decode j)) f) =
  perm_ffun (tperm (fin_decode i) (fin_decode j)) f
```

Equations about finite functions can be proved using the lemma of functional extensionality `ffunP`. Applications of finite functions can be simplified with the `ffunE` lemma.

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

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

The remaining goal can be solved by case analysis and basic equational reasoning.

```
  rewrite permE /=; do!case: eqP; congruence.
```

The `congruence` tactic [19] is a decision procedure for the equational theory of uninterpreted (and constructor) function symbols and implements the congruence closure algorithm [20].

A `swap` function using indices of type `I : finType` can be defined as follows.

```
Definition swap (I : finType) {A : Type} (i j : I) :
  AState {ffun I → A} unit :=
  oswap (fin_encode i) (fin_encode j).
```

We then extract the OCaml implementation.

```
Extraction Inline oswap.
Extraction oswap.
Extraction swap.
```

```
(** val oswap : Finite.coq_type -> ordinal -> ordinal -> 'a1 finfun_of->unit **)

let oswap _ i j =
  (fun f g s -> let r = f s in g r s) ((fun i s -> s.(i)) i) (fun x ->
    (fun f g s -> let r = f s in g r s) ((fun i s -> s.(i)) j) (fun y ->
      (fun f g s -> let r = f s in g r s) ((fun i x s -> s.(i) <- x) i y)
        (fun _ -> (fun i x s -> s.(i) <- x) j x)))
```

```
(** val swap :
    Finite.coq_type -> Finite.sort -> Finite.sort -> 'a1 finfun_of->unit **)

let swap i i0 j =
  let i1 = i.Finite.mixin.Finite.mixin_enc i0 in
  let j0 = i.Finite.mixin.Finite.mixin_enc j in
  (fun f g s -> let r = f s in g r s)
    ((fun i s -> s.(i)) i1) (fun x ->
    (fun f g s -> let r = f s in g r s)
      ((fun i s -> s.(i)) j0) (fun y ->
      (fun f g s -> let r = f s in g r s)
        ((fun i x s -> s.(i) <- x) i1 y) (fun _ ->
        (fun i x s -> s.(i) <- x) j0 x)))
```

Inlining all the redundant $\beta$-redexes of the above `oswap` gives the code that would normally be written by hand to perform the expected operation on a mutable array.

```
let oswap _ i j s =
  let x = s.(i) in
  let y = s.(j) in
  let _ = s.(i) <- y in
  s.(j) <- x
```

Actually, such inlining is among the optimizations performed by the OCaml Flambda optimizer [16, Chapter 21] on its intermediate representation.

### *4.2. Iterations*

Iterations with `for` and `while` loops are very basic constructions in imperative programming in OCaml or other languages. As it is not possible to write general `while` loops without a termination witness in Coq, we only define, verify, and extract `for` like loop combinators on finite ordinals and finite types.

First, we define a `for` like loop combinator with accumulation for descending ordered indices of a finite ordinal `miterate_revord` by using a tail-recursive fixpoint.

```
Fixpoint miterate_revord (S : Type) (g : 'I_n → A → AState S A)
                         (i : nat) (x : A) : i ≤ n → AState S A :=
  match i with
    | 0 ⇒ fun _ ⇒ astate_ret x
    | i'.+1 ⇒
      fun (H : i' < n) ⇒
        mlet y := g (Ordinal H) x in @miterate_revord _ g i' y (ltnW H)
  end.
```

For any `g : 'I_n → A → AState S A` and `x : A`, an iteration `@miterate_revord S g n x (leqnn n)` is equivalent to

```
mlet x_1 := g (@Ordinal n (n - 1) ...) x in
mlet x_2 := g (@Ordinal n (n - 2) ...) x_1 in
⋮
mlet x_n := g (@Ordinal n 0 ...) x_{n-1} in
astate_ret x_n
```

and this equivalence can be easily proved in Coq with the following lemma.

```
Lemma run_miterate_revord
      (S : copyType) (g : 'I_n → A → AState S A) (x0 : A) (s0 : S) :
  run_AState (miterate_revord g x (leqnn n)) s0 =
  foldr (fun i '(x, s) ⇒ run_AState (g i x) s) (x0, s0) (enum 'I_n).
```

We also define loop combinators for ascending and descending ordered indices of both a finite type and a corresponding ordinal by using `miterate_revord`, the decoding function, `cast_ord`, and `rev_ord`, which is a function of type `∀n : nat, 'I_n → 'I_n` such that `nat_of_ord (rev_ord i)` is equal to `n - (i + 1)` for any `i : 'I_n`. Defining a loop combinator for ascending ordered indices of a finite ordinal or a finite type by taking the increasing index as a direct argument requires a non-structurally- or non-tail-recursive fixpoint. We prefer to use `rev_ord` instead and avoid the difficult technique based on non-structural recursion.

```
Definition miterate_both
           (S : Type) (g : 'I_#|T| → T → A → AState S A) (x : A) :
  AState S A :=
  miterate_revord
  (fun i ⇒ let i' := rev_ord i in
            g (cast_ord (raw_cardE _) i') (Finite.decode i')) x.

Definition miterate_revboth
           (S : Type) (g : 'I_#|T| → T → A → AState S A) (x : A) :
  AState S A :=
  miterate_revord
    (fun i ⇒ g (cast_ord (raw_cardE _) i) (Finite.decode i))
    x (leqnn $|T|).
```

By using `miterate_revboth`, a monadic action to reset the array of type `{ffun I → I}` with the identity finite function `[ffun i : I ⇒ i]` can be easily written out.

```
Definition set_identity (I : finType) : AState {ffun I → I} unit :=
  miterate_revboth (fun i i' _ ⇒ astate_SET i i') tt.
```

However, the extracted code is still inefficient despite the use of appropriate inlining instructions. The fixpoint function `miterate_revord` passes its argument `g : 'I_n → A → AState S A` to its recursive call as it is. The extractor cannot recognize and factor out this recursion invariant.

```
(** val set_identity : Finite.coq_type -> Finite.sort finfun_of->unit **)

let set_identity i =
  let rec miterate_revord g i0 x =
    (fun zero succ n -> if n = 0 then zero () else succ (n - 1))
      (fun _ -> (fun a s -> a) x)
      (fun i' ->
      (fun f g s -> let r = f s in g r s) (g i' x) (fun y ->
       miterate_revord g i' y))
      i0
  in miterate_revord (fun i0 _ ->
       (fun i x s -> s.(i) <- x) i0 (i.Finite.mixin.Finite.mixin_dec i0))
       i.Finite.mixin.Finite.mixin_card ()
```

To avoid this source of inefficiency, we redefine `miterate_revord` by factoring out its first argument from the fixpoint term.

```
Definition miterate_revord (S : Type) (g : 'I_n → A → AState S A) :=
  fix rec (i : nat) (x : A) : i ≤ n → AState S A :=
  match i with
    | 0 ⇒ fun _ ⇒ astate_ret x
    | i'.+1 ⇒
      fun (H : i' < n) ⇒
        astate_bind (g (Ordinal H) x) (fun y ⇒ rec i' y (ltnW H))
  end.
```

This is our general strategy to define iterative monadic combinators in an efficient way. An alternative solution for the factoring out would be to use the `Section` mechanism. We would obtain the same result for the efficiency of extracted code. This could actually be a better way to organize the definitions and its accompanying theorems as monadic combinators.

As a result of the above method, extracting `set_identity` gives the following code.

```
let set_identity i =
  let rec rec0 i0 x =
    (fun zero succ n -> if n = 0 then zero () else succ (n - 1))
      (fun _ -> (fun a s -> a) x)
      (fun i' ->
      (fun f g s -> let r = f s in g r s)
        ((fun i x s -> s.(i) <- x) i' (i.Finite.mixin.Finite.mixin_dec i'))
        (fun y -> rec0 i' y))
      i0
  in rec0 i.Finite.mixin.Finite.mixin_card ()
```

Unfortunately, even after inlining

```
  let set_identity i =
    let rec rec0 i0 x =
      if i0 = 0 then fun s ->
        x
      else fun s ->
        let r = s.(i0 - 1) <- i.Finite.mixin.Finite.mixin_dec (i0 - 1) in
        rec0 (i0 - 1) r s
    in rec0 i.Finite.mixin.Finite.mixin_card ()
```

the code is still inefficient because each branch of the `if` expression contains an abstraction `fun s ->` and returns a function. This increases the number of function calls and closure allocations. Generally, this kind of inefficiency cannot be reduced by optimizers for call-by-value functional languages with effects because factoring out such abstractions from inside to outside of a conditional expression or a case analysis construct may change the execution order of expressions.

There is another issue with the original extraction plugin of Coq. The following example `set_identity_ord` instantiates the finite type `I` used for `set_identity` with a concrete finite type instance of `'I_n`.

```
Definition set_identity_ord (n : nat) : AState {ffun 'I_n → 'I_n} unit :=
  set_identity [finType of 'I_n].
```

In this case, a very large record representing the finite type instance of `'I_n` is directly embedded and the accessors for the cardinal number and the encoding/decoding functions are not eliminated in the extracted code.

```
(** val set_identity_ord : int -> ordinal finfun_of->unit **)

let set_identity_ord n =
  let i = (* a very large record representing the [finType of 'I_n] *) in
  Obj.magic
    (let rec rec0 i0 x =
       (fun zero succ n -> if n = 0 then zero () else succ (n - 1))
         (fun _ -> (fun a s -> a) x)
         (fun i' ->
         (fun f g s -> let r = f s in g r s)
           ((fun i x s -> s.(i) <- x) i'
             (i.Finite.mixin.Finite.mixin_dec i')) (fun y ->
           rec0 i' y))
         i0
     in rec0) i.Finite.mixin.Finite.mixin_card ()
```

The next section improves the extraction plugin and solves all these performance problems.

## 5. Optimizations by an improved extraction plugin

The Coq program extraction translates Gallina 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.

We propose two modifications for the extraction plugin to improve the efficiency of extracted programs particularly when they use the array state monad. These modifications concern parts 2 and 1 of the translation respectively. The first modification reduces the construction and destruction costs of (co)inductive objects by inlining. The second one reduces the function call costs by applying $\eta$-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.

### *5.1. Destructing large records*

The MathComp library provides many mathematical structures [14] as canonical structures, e.g., `eqType`, `choiceType`, `countType`, and `finType` which are represented as nested records in extracted programs. For example, our 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 acting on type coercions (`Obj.magic` in OCaml and `unsafeCoerce` in Haskell) are omitted here for the sake of simplicity. The rules we introduce are highlighted by $\star$ and all the other rules are provided by the

original extraction plugin [21, `plugins/extraction/mlutil.ml`][2, Section 4.3.4]. The key simplification rule for unfolding pattern matchings provided by the original extraction plugin is the generalized $\iota$-reduction relation.

**Definition 1** (Generalized $\iota$-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:

$$C_i(t_1, \ldots, t_m) \rightsquigarrow_\iota^{\overline{cl}} \begin{array}{l}\mathsf{let}\ x_{i,1} := t_1\ \mathsf{in} \\ \vdots \\ \mathsf{let}\ x_{i,m} := t_m\ \mathsf{in}\ u_i\end{array} \tag{2}$$

$$\star \qquad t \rightsquigarrow_\iota^{\overline{cl}} t' \Rightarrow \mathsf{let}\ x := u\ \mathsf{in}\ t \rightsquigarrow_\iota^{\overline{cl}} \mathsf{let}\ x := u\ \mathsf{in}\ t' \tag{3}$$

$$t_1 \rightsquigarrow_\iota^{\overline{cl}} t_1' \wedge \cdots \wedge t_m \rightsquigarrow_\iota^{\overline{cl}} t_m' \Rightarrow \begin{array}{l}\mathsf{match}\ t\ \mathsf{with} \\ \mid D_1(\overline{x_1}) \to t_1 \\ \mid \ldots \\ \mid D_m(\overline{x_m}) \to t_m\end{array} \rightsquigarrow_\iota^{\overline{cl}} \begin{array}{l}\mathsf{match}\ t\ \mathsf{with} \\ \mid D_1(\overline{x_1}) \to t_1' \\ \mid \ldots \\ \mid D_m(\overline{x_m}) \to t_m'\end{array} \tag{4}$$

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

$$t \rightsquigarrow_\iota^{\overline{cl}} t' \Rightarrow (\mathsf{match}\ t\ \mathsf{with}\ \overline{cl}) \rightsquigarrow_\iota t' \tag{5}$$

The combination of Rules (2) and (5) provides simple $\iota$-reduction. Rules (3) and (4) allow the traversal of nested match and let expressions of the head term $t$ in Rule (5).

Other simplification rules are listed below.

$$(\lambda x.\, t)\, u \rightsquigarrow \mathsf{let}\ x := u\ \mathsf{in}\ t \tag{6}$$

$$\mathsf{let}\ x := u\ \mathsf{in}\ t \rightsquigarrow t[x := u] \quad \begin{array}{l}(t \text{ or } u \text{ is atomic,} \\ \text{or } x \text{ occurs at most once)}\end{array} \tag{7}$$

$$(\mathsf{let}\ x := t_1\ \mathsf{in}\ t_2)\ u_1 \ldots u_n \rightsquigarrow \mathsf{let}\ x := t_1\ \mathsf{in}\ t_2\ u_1 \ldots u_n \tag{8}$$

$$\begin{array}{l}(\mathsf{match}\ t\ \mathsf{with} \\ \mid C_1(\overline{x_1}) \to t_1 \\ \mid \ldots \\ \mid C_n(\overline{x_n}) \to t_n \\ )\, u_1 \ldots u_m\end{array} \rightsquigarrow \begin{array}{l}\mathsf{match}\ t\ \mathsf{with} \\ \mid C_1(\overline{x_1}) \to t_1\, u_1 \ldots u_m \\ \mid \ldots \\ \mid C_n(\overline{x_n}) \to t_n\, u_1 \ldots u_m\end{array} \tag{9}$$

$$\begin{array}{l}\mathsf{match}\ t\ \mathsf{with} \\ \mid C_1(\overline{x_1}) \to \lambda y_1 \ldots y_m.\, t_1 \\ \mid \ldots \\ \mid C_n(\overline{x_n}) \to \lambda y_1 \ldots y_m.\, t_n\end{array} \rightsquigarrow \begin{array}{l}\lambda y_1 \ldots y_m.\, \mathsf{match}\ t\ \mathsf{with} \\ \qquad \mid C_1(\overline{x_1}) \to t_1 \\ \qquad \mid \ldots \\ \qquad \mid C_n(\overline{x_n}) \to t_n\end{array} \tag{10}$$

$$\star \quad \mathsf{let}\ x := (\mathsf{let}\ y := t_1\ \mathsf{in}\ t_2)\ \mathsf{in}\ t_3 \rightsquigarrow \begin{array}{l}\mathsf{let}\ y := t_1\ \mathsf{in} \\ \mathsf{let}\ x := t_2\ \mathsf{in} \\ t_3\end{array} \tag{11}$$

$$\star \quad \mathsf{let}\ x := C(t_1, \ldots, t_n)\ \mathsf{in}\ u \rightsquigarrow \begin{array}{l}\mathsf{let}\ y_1 := t_1\ \mathsf{in} \\ \vdots \\ \mathsf{let}\ y_n := t_n\ \mathsf{in} \\ \mathsf{let}\ x := C(y_1, \ldots, y_n)\ \mathsf{in} \\ u'\end{array} \tag{12}$$

where $u'$ in Rule (12) is obtained by replacing all 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 $\iota$-reduced term of ($\mathsf{match}\ C(y_1, \ldots, y_n)\ \mathsf{with} \cdots \mid C(z_1, \ldots, z_n) \to t \mid \ldots$).

Some simplification rules shown above are not safe for OCaml programs in general. Rules (7), (9) and (10) may change the execution order of program fragments and skip executing some fragments, 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 them as a MiniML optimizer.

Rules (3), (11), and (12) are new rules. All the other rules are present in the original extraction plugin. Rule (12) recursively destructs nested records, and Rule (11) is needed to apply Rule (12) for terms of the form ($\mathsf{let}\ x := (\mathsf{let}\ y := t\ \mathsf{in}\ C(t_1, \ldots, t_n))\ \mathsf{in}\ \ldots$).

*5.2. $\eta$-expansion on case analysis*

Match expressions returning a function are commonly used in monadic programming using monads with function type (e.g., state monad, reader monad, continuation monad, other monads constructed by monad transformers of them) as we saw in Section 4.2 and dependently typed programming. However, they increase the number of function calls and closure allocations. This decreases the performance of programs. Let us consider the following monadic program that branches depending on whether the integer state is even or odd:

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

As `if` expressions in Coq are syntax sugar for match expressions, Unfolding *get* and $\gg\!=$ in the above program leads to a match expression returning a function:

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

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

```
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 Rules (9) and (10) respectively, and these rules can be applied in many other cases. However, these rules are not complete because of their syntactic restriction; that is, match expressions where these rules are applied must either have arguments or have $\lambda$-abstractions in all branches. As some type information on terms are necessary to implement this full $\eta$-expansion and no type information is available at the MiniML level (MiniML is a type-free language), we apply the full $\eta$-expansion on all match expressions in the process of extraction from Gallina to MiniML. Rule (9) can then be applied as soon as the type of the expression is a function type.

To perform $\eta$-expansion, we need extra typing information. In particular, the type `AState` is extracted toward a function type with one argument. We say that its extracted type has arity one. We provide a new Vernacular command `Extract Type Arity` to declare the arity of an inductive type which is realized by the `Extract Inductive` command, Declared arities are used for full $\eta$-expansion as described above. The command

```
Extract Type Arity AState 1.
```

helps the Coq system recognizing that the arity of `AState` is 1 in OCaml. The extracted code of the function `set_identity_ord` described in Section 4.2 is then

```
(** val set_identity_ord : int -> ordinal finfun_of->unit **)

let set_identity_ord n =
  Obj.magic (fun x ->
    let rec rec0 i x0 x1 =
      (fun zero succ n -> if n = 0 then zero () else succ (n - 1))
        (fun _ -> (fun a s -> a) x0 x1)
        (fun i' ->
        (fun f g s -> let r = f s in g r s)
          ((fun i x s -> s.(i) <- x) i' (Obj.magic i')) (fun y x2 ->
          rec0 i' y x2) x1)
        i
    in rec0 n () x)
```

As it can be seen in the following simplified definition of `set_identity_ord`, no branch of the `if` expression returns a function, the finite type instance of `'I_n` has disappeared, and the accessors for the cardinal number and the encoding/decoding functions are inlined (the encoding/decoding functions for finite ordinal types are the identity function); therefore, the inefficiency issues we mentioned have disappeared.

```
let set_identity_ord n =
  Obj.magic (fun x ->
    let rec rec0 i x0 x1 =
      if i = 0 then
        x0
      else
        let i' = i - 1 in
        let r = x1.(i') <- Obj.magic i' (* = mixin_dec i' *) in
        rec0 i' r x1
    in rec0 n (* = mixin_card *) () x)
```

## 6. Case studies

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

The benchmark programs were compiled by OCaml 4.07.0+flambda+no-flat-float-array. The optimization flags are `-O3`, `-rounds 10`, `-remove-unused-arguments`, and `-unbox-closures`. They are executed on a Intel Core i5-7260U CPU @ 2.20GHz equipped with 32 GB of RAM. Full major garbage collection and heap compaction (`Gc.compact`) are invoked before each measurement. All the benchmark results are medians of 5 different measurements with the same parameters.

### *6.1. The union–find data structure*

We implemented and verified the union–find data structure with the path compression and the weighted union rule [22, 23]. The monadic definitions of union and find are shown in the Listing 1.

Listing 1: The monadic implementation of the union–find data structure

```
Fixpoint mfind_rec (R : eqType) (Ridx : R) (T : finType) (n : nat) (x : T) :
    AState {ffun T → T + R} (T * R) :=
  if n is n'.+1
  then mlet x' := astate_get x in
       match x' with
         | inl x'' ⇒
           mlet '((rep, _) as r) := mfind_rec n' x'' in
           astate_set x (inl rep);; astate_ret r
         | inr a ⇒ astate_ret (x, a)
       end
  else astate_ret (x, Ridx).

Definition mfind (R : eqType) (Ridx : R) (T : finType) := mfind_rec Ridx $|T|.

Definition munion
    (R : eqType) (Ridx : R) (Rop : Monoid.com_law Ridx) (cmp : rel R)
    (T : finType) (x y : T) : AState {ffun T → T + R} unit :=
  mlet '(xr, xv) := mfind Ridx x in
  mlet '(yr, yv) := mfind Ridx y in
  if xr == yr then astate_ret tt else
  if cmp xv yv
    then astate_set xr (inl yr);; astate_set yr (inr (Rop xv yv))
    else astate_set yr (inl xr);; astate_set xr (inr (Rop xv yv)).
```

Listing 2: Key definitions for the reasoning of the union–find data structure

```
Variable (R : eqType) (Ridx : R) (T : finType) (g : {ffun T → T + R}).

Definition repr (x : T) : bool := if g x is inr _ then true else false.
Definition succ (x : T) : T := if g x is inl x' then x' else x.
Definition cdatum (x : T) : R := if g x is inr a then a else Ridx.

Definition path := path [rel x y | g x == inl y].
Definition connect := fconnect succ.

Definition find : T → T := iter #|T| succ.
Definition findeq (x y : T) : bool := find x == find y.

Definition lacycle (x : T) : bool := repr (find x).
Definition acycle := ∀x : T, lacycle x.
```

The union–find forests are represented in our implementation by `T` indexed arrays of `T + R` where `T : finType`. For a union–find forest `g : {ffun T → T + R}` and an element `x : T`, `g x = inl x'` means that `x` is not a representative and `x'` is the successor of `x` where `x' : T`, and `g x = inr v` means that `x` is a representative and `v` is the data attached to the equivalence class of `x` where `v : R`. Graphs representing the union–find forests should not have cyclic paths and therefore lengths of paths should be shorter than `#|T|` (or `$|T|`). This number is used as the loop upper-bound for the find operation.

The type of attached data `R` should be a commutative monoid with an identity `Ridx : R` and a binary operation `Rop : Monoid.com_law Ridx` which is coercible to a function of type `R → R → R`. When two classes are joined by the union operation, the attached data of them are joined by `Rop` as well. Attached data satisfying commutative monoid axioms are ubiquitously used in many applications of the union–find data structure, e.g., weights used for the weighted union operation, and sets of inhabitants used for the congruence closure. We have obtained weighted union–find by instantiating this abstract union–find with weights. Two different kind of attached data could be combined as their product. Therefore this abstraction is quite useful to implement such applications.

To state some properties on paths and connectivity on union–find forests, our formalization uses the path and fingraph libraries and extra key definitions (Listing 2). For a union–find forest `g : {ffun T → T + R}`, `repr x` is true if and only if `x` is a representative. `succ x` is the successor of `x` except when it is a representative. It is `x` itself in that case. `cdatum x` is the attached data of `x` except when `x` is not a representative. It is `Ridx` in that case. The connectivity on `g` is defined as the connectivity under iteration of the successor (`fconnect succ`); however, the paths are defined by using another binary relation `[rel x y | g x == inl y]` to avoid iteration after representative elements. To reflect the connectivity to the existence of a path, we need to provide a lemma like `fingraph.connectP` as follows.

```
Lemma connectP (x y : T) :
  reflect (exists2 p : seq T, path x p & y = last x p) (connect x y).
```

Containing no cyclic paths is an essential invariant of the union–find forests. If there is no cyclic path, every path is shorter than `#|T|`; thus, the representative (`find`) of a class containing `x : T` can be obtained by `#|T|` times application of the successor on `x`. We have defined that `x : T` is locally acyclic (`lacycle x`) if and only if `find x` is a representative, which is equivalent to that paths from `x` have finite length and have no cycles. The global acyclic property of union–find forests (`acycle`) is defined as `∀ x : T, lacycle x`. The local acyclic property can be carried along paths by using the following lemmas and gives us a way to reason about acyclicity of an equivalence class in union–find forests.

```
Lemma lacycle_succ x : lacycle (succ x) = lacycle x.
Lemma lacycle_connect x y : connect x y → lacycle x = lacycle y.
```

We have proved the correctness of the weighted union–find with path compression in 487 lines of code (193 lines of specifications and 294 lines of proofs) including definitions thanks to the above techniques.

The benchmark results are shown in the Figure 1. Here we compare the execution times of the OCaml program extracted from above formalization (optimized and

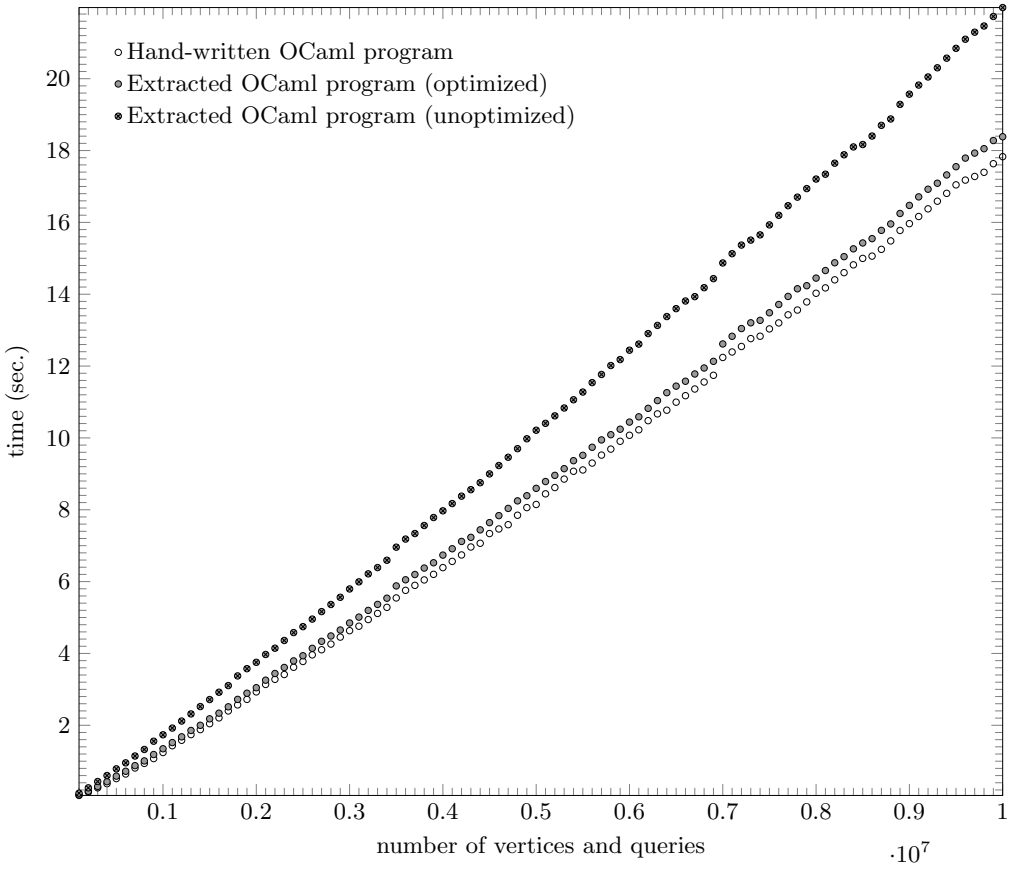


Figure 1: Benchmark results of the union–find data structure

unoptimized version) and a handwritten OCaml implementation of same algorithm. The unoptimized version is obtained by disabling the new optimization mechanisms introduced in Section 5, but is compiled with the same OCaml compiler and optimization flags. The procedure to be measured here is the sequence of *union* $n$ times and *find* $n$ times on $n$ vertices union–find data structure, where all parameters of *union*s and *find*s are randomly selected. The time complexity of this procedure is $\Theta(n\alpha(n,n))$ where $\alpha$ is a functional inverse of Ackermann's function. The results indicate that the optimized Coq implementation is slightly slower than the OCaml implementation and approximately 1.2 times faster than the unoptimized Coq implementation, and the execution times of all implementations increase slightly more than linearly.

### *6.2. The quicksort algorithm*

We implemented and verified the quicksort algorithm by using the array state monad. The monadic definitions of partitioning (`partition`) and quicksort (`quicksort_rec` and `quicksort`) are shown in the Listing 3. We have defined them with carrying inequality constraints of array indices and gave an explicit proof of $m \leq n$ for every subtraction $n-m$ to avoid case analyses on subtractions in the extracted program. On the verification side, most inequality proofs had been done by the `lia` (linear integer arithmetic) tactic [24] and the key properties are elegantly written out by using the theory of permutations.

Let us explain our reasoning techniques used for the quicksort algorithm by taking the partitioning function as an example. The partitioning function `partition` for `I` indexed arrays of `A` and comparison function `cmp : A → A → bool`[4] takes a pivot of type `A` 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.[5]

```
Variant partition_spec (pivot : A) (i j : 'I_#|I|.+1) (arr : {ffun I → A}) :
  {k : 'I_#|I|.+1 | i ≤ k ≤ j} * {ffun I → A} → Prop :=
  PartitionSpec (p : {perm I}) (k : 'I_#|I|.+1) (Hk : i ≤ k ≤ j) :
  let arr' := perm_ffun p arr in
  (*1*) perm_on [set ix | i ≤ fin_encode ix < j] p →
  (*2*) (∀ix : 'I_#|I|, i ≤ ix < j →
                        cmp pivot (arr' (fin_decode ix)) = (k ≤ ix)) →
        @partition_spec pivot i j arr (exist _ k Hk, arr').

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) arr).
```

The `run_partition` lemma can be applied on goals that contains some calls to `partition` without giving concrete parameters by using the simple idiom `case: run_partition`, and one directly obtains the properties of `partition` described below. It follows a

[4] We interpret `cmp x y` as "`x` is less than or equal to `y`" in our explanation. Nevertheless, `cmp` can be any total and transitive relation without loss of generality.

[5] The `Variant` command is identical to the `Inductive` command except that it disallows recursive definition of types [5]. It is commonly used in MathComp to define non-recursive data types and type families without generating induction schemes. Historically, the `CoInductive` command was used for this purpose.

Listing 3: The monadic implementation of the quicksort algorithm

```
Variable (I : finType) (A : Type) (cmp : A → A → bool).

Definition partition (pivot : A) :
  ∀(i j : 'I_#|I|.+1),
    i ≤ j → AState {ffun I → A} {k : 'I_#|I|.+1 | i ≤ k ≤ j} :=
  Fix
    (@well_founded_ordgt #|I|.+1)
    (fun i ⇒ ∀j : 'I_#|I|.+1,
        i ≤ j → AState {ffun I → A} {k : 'I_#|I|.+1 | i ≤ k ≤ j})
    (fun (i : 'I_#|I|.+1) rec (j : 'I_#|I|.+1) (Hij : i ≤ j) ⇒
       mlet '(exist i' Hi) := up_search (cmp pivot) (i := i) (j := j) Hij in
       mlet '(exist j' Hj) :=
         down_search (xpredC (cmp pivot)) (i := j) (j := i) Hij in
       match i'.+1 < j' as cij return i'.+1 < j' = cij → _ with
         | true ⇒ fun Hij : i'.+1 < j' ⇒
           oswap (ltnidx_l (ltnW Hij)) (* = i' : 'I_#|I| *)
                 (ltnidx_rp (ltnm0m Hij)) (* = j' - 1 : 'I_#|I| *);;
           mlet '(exist k Hk) :=
             rec (ltnidx_ls (ltnW Hij)) (* = i' + 1 : 'I_#|I|.+1 *)
                 ltac: (by case/andP: Hi; rewrite ltnS)
                 (ord_pred' (i := j') (ltnm0m Hij)) (* = j' - 1 : 'I_#|I|.+1 *)
                 ltac: (by case: (j') Hij ⇒ -[]) in
           astate_ret (exist (fun k : 'I_ _ ⇒ i ≤ k ≤ j) k
             ltac:(move: Hi Hj Hk ⇒ /andP [Hi _] /andP [_ Hj] /andP [Hkl Hkr];
                   rewrite (leq_trans Hi (ltnW Hkl)) (leq_trans Hkr) //
                           (leq_trans (leq_pred _)) //))
         | false ⇒ fun _ ⇒
           astate_ret (exist (fun k : 'I_ _ ⇒ i ≤ k ≤ j) i' Hi)
       end (erefl (i'.+1 < j'))).

Definition quicksort_rec :
  'I_#|I|.+1 → 'I_#|I|.+1 → AState {ffun I → A} unit :=
  Fix
  (@well_founded_ordgt #|I|.+1)
  (fun _ ⇒ 'I_#|I|.+1 → AState {ffun I → A} unit)
  (fun (i : 'I_#|I|.+1) recl ⇒
     Fix
     (@well_founded_ordlt #|I|.+1)
     (fun _ ⇒ AState {ffun I → A} unit)
     (fun (j : 'I_#|I|.+1) recr ⇒
        match i.+1 < j as cij return i.+1 < j = cij → _ with
          | true ⇒ fun Hij : i.+1 < j ⇒
            let i' := ltnidx_l (ltnW Hij) (* = i : 'I_#|I| *) in
            let si := ltnidx_ls (ltnW Hij) (* = i + 1 : 'I_#|I|.+1 *) in
            mlet pivot := astate_GET i' in
            mlet '(exist k Hk) := @partition pivot si j (ltnW Hij) in
            let Hk' : 0 < k := ltac:(by case/andP: Hk ⇒ /ltnm0m) in
            let pk := ltnidx_rp (j := k) Hk' (* = k - 1 : 'I_#|I| *) in
            mlet x := astate_GET pk in
            astate_SET i' x;; astate_SET pk pivot;;
            recr (ord_pred' (i := k) Hk') (* = k - 1 : 'I_#|I|.+1 *)
                 ltac:(by case/andP: (Hk); rewrite /= (ltn_predK Hk'));;
            recl k ltac:(by case/andP: (Hk)) j
          | false ⇒ fun _ ⇒ astate_ret tt
        end (erefl (i.+1 < j)))).

Definition quicksort : AState {ffun I → A} unit :=
  quicksort_rec ord0 (@Ordinal #|I|.+1 $|I| ltac:(by rewrite raw_cardE)).
```

Ssreflect convention [8, Section 4.2.1], where the specification is described by a type family `partition_spec`.

In the specification `partition_spec`, the finite function `arr` describes the initial state, and the permutation `p` and the index `k` describe permutation performed by the partition, and the partition position respectively. The final state `arr'` is given by `perm_ffun p arr` which means the finite function `arr` permuted by `p`. Properties of the partition are given as arguments of `PartitionSpec` and numbered `(*1*)` and `(*2*)` in the above code. The former proposition indicates that the partition only replaces value in the range from `i` to `j - 1`. The latter proposition indicates that values in the range from `i` to `k - 1` are less than the pivot and values in the range from `k` to `j - 1` are greater than or equal to the pivot.

Of particular interest is that the former property can be written in the form of `perm_on a p` which means that the permutation `p` only moves elements of `a` around. `perm_on` is originally defined to obtain some algebraic results in the MathComp library, but is also helpful for reasoning on sorting algorithms.

We have proved the correctness of the quicksort algorithm in 304 lines of code (178 lines of specifications and 126 lines of proofs) including partitioning and the upward/-downward search thanks to the above techniques.

The benchmark results are shown in the Figure 2. Here we compare the execution times of the following implementations of sorting algorithms for randomly generated arrays or lists of integers: 1. `Array.stable_sort` and 2. `Array.sort` taken from the OCaml standard library,[6] 3. a handwritten OCaml implementation of the quicksort algorithm, 4. optimized and 5. unoptimized OCaml programs extracted from the above formalization, and 6. an OCaml program extracted from a Coq implementation of the bottom-up mergesort algorithm for lists. The results indicate that the implementation 5 (quicksort in Coq, unoptimized) is the slowest, the implementation 4 (quicksort in Coq, optimized) is approximately 10 and 2 times faster than the other extracted implementations 5 and 6 respectively and also approximately 1.1 times faster than `Array.sort` (provided in the OCaml libraries), and the other OCaml implementations 1 and 3 are approximately 1.5 and 1.4 times faster than the implementation 4 respectively.

## 7. Related work

There are many recent works to support writing, reasoning about, and generating programs with side effects in proof assistants, e.g., Coq [25, 26, 27, 28], Isabelle/HOL [29], Idris [30], and F$^\star$ [31]. Ynot [25] is a representative (but no longer maintained) Coq library for verification and extraction of higher-order imperative programs.

Among these works, Ynot is based on an axiomatic extension for Coq called Hoare Type Theory (HTT) [32] and supports various kind of side-effects (e.g., accessing reference cells, non-termination, throwing/catching exceptions, and I/O), reasoning with a form separation logic [33, 34], and automation facility [35] for it. Ynot provides many example implementations of and formal proofs for data structures and algorithms including the union–find data structure. CFML [26] is a tool to reason about imperative OCaml programs in Coq. CFML parses a given OCaml program and produces a characteristic

[6] Their current implementations are mergesort and heapsort respectively.

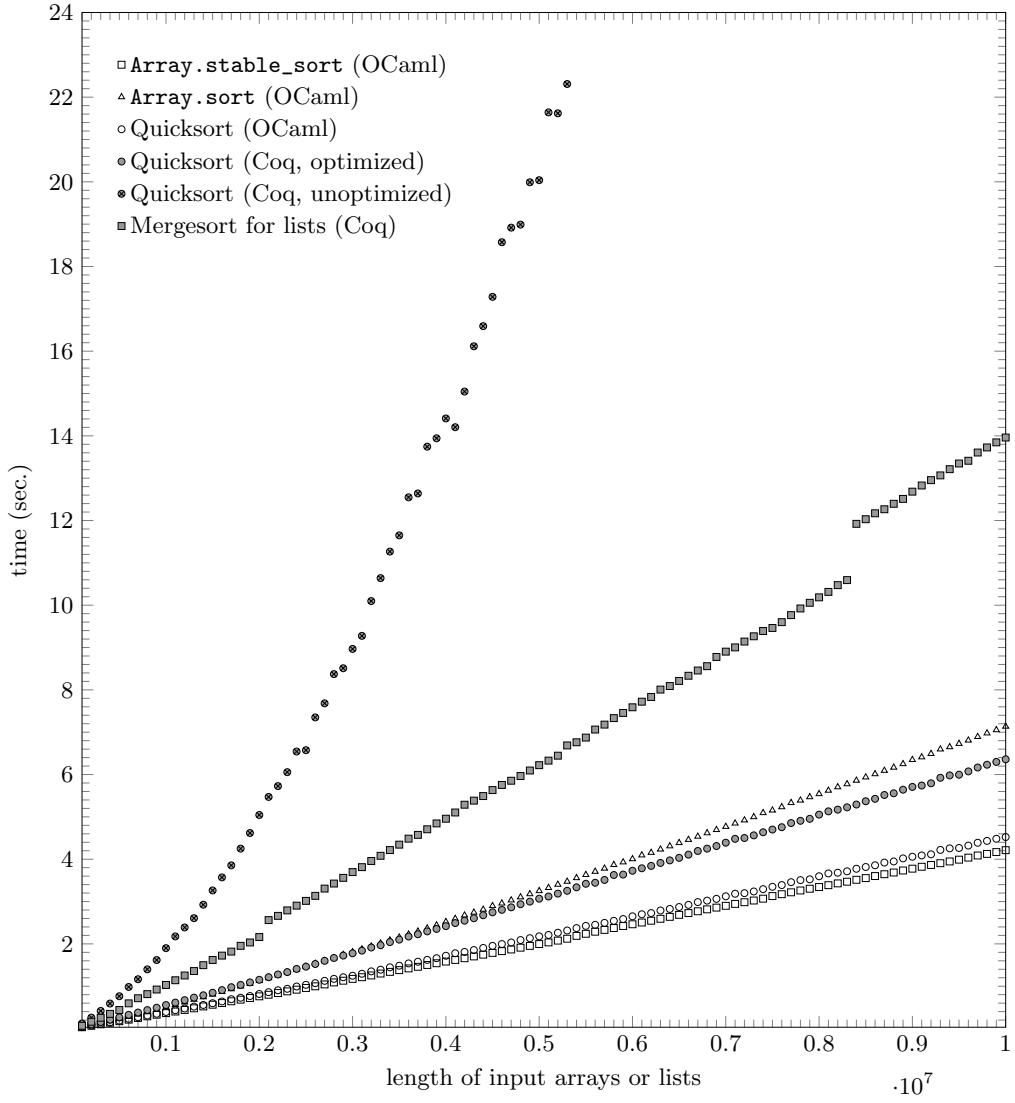


Figure 2: Benchmark results of the quicksort and mergesort

formula which precisely characterizes the given program; thus, CFML has no execution overhead like ours, but cannot use any language features of Gallina to write imperative programs. CFML also provides a Coq library to enable separation-logic-based reasoning using those characteristic formulae. Recently, CFML has been extended with time credits to support complexity reasoning; as a result, they have verified the functional correctness and the worst-case amortized time complexity of union–find [36]. FCSL [27] and Iris [28] are modern Coq framework for verification of fine-grained concurrent programs. FCSL defines a denotational semantics of semantic objects called action trees [37] and defines a language of concurrent programs on top of them by shallow embedding; thus its language supports the full expressiveness of Coq including dependent types and pattern matching. Iris is parameterized by the language of programs that one wishes to reason about, but is typically instantiated by a language defined by deep embedding, and thus does not immediately support dependent types and pattern matching. Both frameworks are developed without declaring axioms in contrast to Ynot but do not provide any code generation method producing those fine-grained concurrent programs.

Ynot and CFML use almost the same optimization strategies as our development in their implementations of union–find: the union by rank and the path compression. Our development implements the weighted union rule by instantiating the attached data of the equivalence classes instead of the union by rank (Section 6.1). The Ynot's development consists of 855 lines of code (480 lines of specifications and 375 lines of proofs), and is 75% larger than our development. The CFML's development consists of 4,296 lines of the mathematical analysis part, 355 lines of specifications, and 421 lines of proofs. In CFML's specification and verification style, the functional correctness and the complexity analysis have been put together inside Hoare triples; thus, it is difficult to strip the complexity analysis part and to provide a fair and informative comparison with our development on sizes of proofs. We and Ynot use fixed-size arrays to represent union–find forests; thus we cannot introduce a new equivalence class. CFML uses references to represent them; so they can introduce a new equivalence class by making a fresh reference.

Our development cannot represent and reason about effects other than reading and writing mutable arrays; however, this lack of expressiveness has some benefits on the verification side. In general, separation-logic-based reasoning on an imperative data structure requires to define an abstract model of the data structure to reason about it. It also require to establish a correspondence between the model and objects on the heap. For example, Ynot defines an inductive data type representing union–find forests, and CFML uses `Prop` relations of vertices as a model and defines acyclicity on them. In our case studies, we did not need to define this model and to establish a correspondence: we use finite functions as a model of union–find forests, and define acyclicity (`lacycle`) on them as a Boolean function directly. Thus, our specification and verification style equates arrays representing an imperative data structure and its model. This simplifies our proofs.

## 8. Conclusion and future work

We have established a lightweight method for verification and extraction of efficient effectful programs using mutable arrays in Coq. This method consists of the following three parts:

1. a modified MathComp library which establishes an abstraction for immutable arrays supporting arbitrary finite types as indices and having efficient underlying computations for most of the typical finite types,
2. a state monad specialized for mutable array programming (the array state monad) which enables a simple reasoning method and safe extraction of programs involving mutable arrays, and
3. an improved extraction plugin for Coq which optimizes programs extracted from formal developments using our library and is also effective for mathematical structures provided by the MathComp library and monadic programs.

As a result, we have verified and extracted reasonably fast programs from monadic implementation of the union–find data structure and the quicksort algorithm in Coq. The following sections discuss some issues of this work and their possible solutions.

### *8.1. Trusted computing base (TCB)*

This section takes a look at the trusted computing base (TCB) of our method. We have to trust the following elements if we want to trust programs obtained by our method.

**TCB of the Coq system** Our monadic programming and verification method is constructed on top of the Coq system. Thus we have to trust the TCB of the Coq system itself[7]; namely, consistency of the logic behind of Coq, its implementation (kernel), the OCaml compiler, and the underlying execution environment. The MathComp library, our development of the array state monad, and its case studies do not declare and use any axiom we need to trust.

**The extraction plugin of Coq** The extraction mechanism which produces executable OCaml code from Gallina terms is not a part of the Coq kernel and is implemented as a plugin.

The extraction algorithm of Coq has been proved correct on paper [2], but its implementation has not been formally verified; thus, using extraction makes the TCB of the code generation method larger. Some recent works on code generation from interactive theorem provers use (verified or proof-producing) binary code extraction methods [38, 39] rather than unverified classical code generation methods to reduce their TCB.

In our work, we could not use any existing verified code generation method because we use the expressiveness of Gallina extensively and require the full extraction from Gallina. One could be able to use the extraction procedure of the MetaCoq project [40] to make our extraction method more reliable; however, the current development of MetaCoq has some gaps with the extraction plugin we use:

- extracted terms in MetaCoq are completely untyped and have no type coercions to make the Hindley–Milner type inference does work for them, and
- the extraction procedure of MetaCoq has no simplification mechanism like in Section 5.1.

[7] https://github.com/coq/coq/wiki/Presentation#what-do-i-have-to-trust-when-i-see-a-proof-checked-by-coq

Therefore, it would be worth to define MiniML with type coercions, its static and dynamic semantics, and its simplification procedure, and prove some of the following desirable properties.

- The simplification procedure terminates, returns a normal form with respect to the simplification rules, and preserves typing and observation.
- The size of an output term of the simplification procedure is polynomially bounded by the size of the input.[8]

We have sketched correctness proofs of the simplification *rules* in Appendix A, but we expect that formally verifying the correctness properties of the simplification *procedure* stated above is more challenging task.

**Realization directives for the extraction plugin** We have been using a lot of commands `Extract Inductive` and `Extract Constant` to replace some Coq types and definitions through the extraction with OCaml types and terms so that can be handled and executed efficiently. In particular, the key idea of our method —the array state monad and its effectful program extraction— is implemented in this way. Its correctness is non trivial and left as a conjecture.

From a different point of view, we gave an effectful interpretation for the array state monad by extraction and we can also interpret the array state monad as it is defined in Coq without introducing effects. We expect that one could prove an equivalence of those interpretation by using a logical relation, e.g., by extending [41]. By stating and proving this conjecture as a metatheorem, we could breakdown and reduce this TCB.

**Monadic programs should not contain some internal constants** The constants `AState_rect` and `run_AState_raw` are introduced to establish our extraction and encapsulation mechanism in Section 3 and must not be directly used for array state monad programming. This can be seen both as an invariant that users must respect and as a TCB of our method.

We have tried to seal those definitions by module types and modules to prohibit their undesirable uses, and failed due to a technical issue on modules and extraction [42].

### *8.2. Expressiveness of the array state monad*

The array state monad could be extended in several ways to obtain more expressiveness.

The finfun library has been generalized to support dependently typed functions [43] in the MathComp 1.8.0 library. Currently, our development is based on MathComp 1.7.0; thus, it supports only simply typed finite functions. However, one may apply our modification method of Section 2 for this generalized finite functions to use them as a representation of dependently typed arrays whose types of elements are depending on

[8] In the current implementation, we have a counterexample of this property which produces exponentially bigger terms. This is caused by the generalized $\iota$-reduction and a issue of the Coq system itself. https://github.com/coq/coq/issues/7830

their indices. As an application of this generalization, union–find forests for a union–find with *Explain* and a proof-producing congruence closure algorithm [44] can be represented as finite functions of type `{ffun ∀x : T, {y : T & Path x y} + R}` where `T` is vertices, `R` is attached data (see Section 6.1), and `Path : T → T → Type` is a type of dependently typed, reflexive, transitive, and symmetric path objects. Thus one may write those algorithms in Coq and extract reasonably fast implementations from them along this idea.

One may extend the array state monad to an indexed state monad to allow transposing arrays, e.g., from `(f, g)` to `(g, f)`, from `((f, g), h)` to `(f, (g, h))`, and vice versa. We have tried to obtain more expressiveness and modularity with this extension; however, it made necessary to return a new state in each monadic action and we encountered a serious performance issue due to this. As a possible solution, one could introduce a new type argument for the array state monad to trace those transpositions (e.g., `fun '(f, g) ⇒ (g, f)`) and do not return a new state in each monadic action. This slightly restrictive indexed array state monad with the tracing mechanism would be enough to obtain the expressiveness and modularity we need and may help the OCaml compiler to produce efficiently executable programs.

As the benchmark results of Section 6 indicate, the extracted programs are (slightly) slower than the hand-written OCaml programs. In the first case study —the union–find data structure—, one of the reasons for this performance issue might be the fuel parameter `n : nat` in `mfind_rec`. To erase this fuel parameter, we need to find and reason about a decreasing parameter in the state; however, it is not possible with the current representation of the array state monad. To enable this kind of reasoning, one may interest to extend our method with the Hoare state monad [45].

### *8.3. Efficiency issues of programs obtained by the extraction plugin*

As we mentioned in the last sentence of Section 3.4, packed classes introduce the universal type `Obj.t` and many type coercions `Obj.magic` in extracted programs. Erasing type information by `Obj.magic` sometimes makes executions of OCaml programs slower. For example, `Obj.magic (=) 0 i` is slower than `(=) 0 i` where `i` is a variable of type `int` because `(=)` is not monomorphized thus become the polymorphic comparison `compare_val` in the former one. The following small example reproduces the extraction side of this issue.

```
Record type := Pack { sort : Type }.
Definition nat_type := Pack nat.
Definition nat_Type := sort nat_type.
Recursive Extraction nat_Type.
```

```
type __ = Obj.t
type sort = __
type nat_Type = sort
```

The optimizations we proposed in Section 5.1 do solve some inefficiencies introduced by packed classes; however, they do not eliminate those type coercions. In order to eliminate all the record constructions and projections involving packed classes, it is necessary to inline all the definitions involving packed classes. Thus our extraction method may then not be easily applicable for huge development using the MathComp library.

As an alternative solution to solve the above issues, one could write a transformation of Gallina terms to eliminate record constructions and projections by the record's eta expansion.

*Acknowledgments*

We would like to thank Yukiyoshi Kameyama and the anonymous referees for their valuable comments on earlier versions of this article. We also thank all the participants of the JSPS-CNRS bilateral program "FoRmal tools for IoT sEcurity" (PRC2199) for fruitful discussions. This work was supported by JSPS Research Fellowships for Young Scientists and JSPS KAKENHI Grant Number 17J01683.

## References

[1] P. Letouzey, A new extraction for Coq, in: TYPES '03, Vol. 2646 of LNCS, Springer, 2003, pp. 200–219.
[2] P. Letouzey, Programmation fonctionnelle certifiée – l'extraction de programmes dans l'assistant Coq, Ph.D. thesis, Université Paris-Sud (2004).
[3] P. Letouzey, Extraction in Coq: An overview, in: CiE '08, Vol. 5028 of LNCS, Springer, 2008, pp. 359–369.
[4] C. Paulin-Mohring, Extracting $F_\omega$'s programs from proofs in the Calculus of Constructions, in: POPL '89, ACM, 1989, pp. 89–104.
[5] The Coq Development Team, The Coq Proof Assistant Reference Manual, https://coq.inria.fr/distrib/V8.9.0/refman/ (2019).
[6] X. Leroy, A formally verified compiler back-end, J. Autom. Reason. 43 (4) (2009) 363.
[7] The Mathematical Components project, The mathematical components repository, https://github.com/math-comp/math-comp.
[8] A. Mahboubi, E. Tassi, Mathematical components, https://math-comp.github.io/mcb/book.pdf (2018).
[9] K. Sakaguchi, Y. Kameyama, Efficient finite-domain function library for the Coq proof assistant, IPSJ Trans. on Prog. 10 (1) (2017) 14–28.
[10] J. Launchbury, S. L. Peyton Jones, Lazy functional state threads, in: PLDI '94, ACM, 1994, pp. 24–35.
[11] K. Sakaguchi, Program extraction for mutable arrays, in: FLOPS '18, Vol. 10818 of LNCS, Springer, 2018, pp. 51–67.
[12] G. Gonthier, Formal proof—the four-color theorem, Notices of the AMS 55 (11) (2008) 1382–1393.
[13] G. Gonthier, A. Asperti, J. Avigad, Y. Bertot, C. Cohen, F. Garillot, S. Le Roux, A. Mahboubi, R. O'Connor, S. Ould Biha, I. Pasca, L. Rideau, A. Solovyev, E. Tassi, L. Théry, A machine-checked proof of the odd order theorem, in: ITP '13, Vol. 7998 of LNCS, Springer, 2013, pp. 163–179.
[14] F. Garillot, G. Gonthier, A. Mahboubi, L. Rideau, Packaging mathematical structures, in: TPHOLs '09, Vol. 5674 of LNCS, Springer, 2009, pp. 327–342.
[15] A. Mahboubi, E. Tassi, Canonical structures for the working Coq user, in: ITP '13, Vol. 7998 of LNCS, Springer, 2013, pp. 19–34.
[16] X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, J. Vouillon, The OCaml system release 4.07, http://caml.inria.fr/pub/distrib/ocaml-4.07/ocaml-4.07-refman.pdf (2018).
[17] P. Wadler, Monads for functional programming, in: AFP '95, Vol. 925 of LNCS, Springer, 1995, pp. 24–52.
[18] C.-k. Lin, Programming monads operationally with Unimo, in: ICFP '06, ACM, 2006, pp. 274–285.
[19] P. Corbineau, Deciding equality in the constructor theory, in: TYPES '06, Vol. 4502 of LNCS, Springer, 2007, pp. 78–92.
[20] G. Nelson, D. C. Oppen, Fast decision procedures based on congruence closure, J. ACM 27 (2) (1980) 356–364.
[21] The Coq Development Team, The Coq proof assistant (github repository), https://github.com/coq/coq.
[22] R. E. Tarjan, Efficiency of a good but not linear set union algorithm, J. ACM 22 (2) (1975) 215–225.
[23] R. E. Tarjan, J. van Leeuwen, Worst-case analysis of set union algorithms, J. ACM 31 (2) (1984) 245–281.

[24] F. Besson, Fast reflexive arithmetic tactics the linear case and beyond, in: TYPES '06, Vol. 4502 of LNCS, Springer, 2006, pp. 48–62.
[25] A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau, L. Birkedal, Ynot: Dependent types for imperative programs, in: ICFP '08, ACM, 2008, pp. 229–240.
[26] A. Charguéraud, Characteristic formulae for the verification of imperative programs, in: ICFP '11, ACM, 2011, pp. 418–430.
[27] I. Sergey, A. Nanevski, A. Banerjee, Mechanized verification of fine-grained concurrent programs, in: PLDI '15, ACM, 2015, pp. 77–87.
[28] R. Jung, D. Swasey, F. Sieczkowski, K. Svendsen, A. Turon, L. Birkedal, D. Dreyer, Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning, in: POPL '15, ACM, 2015, pp. 637–650.
[29] L. Bulwahn, A. Krauss, F. Haftmann, L. Erkök, J. Matthews, Imperative functional programming with Isabelle/HOL, in: TPHOLs '08, Vol. 5170 of LNCS, Springer, 2008, pp. 134–149.
[30] E. Brady, Programming and reasoning with algebraic effects and dependent types, in: ICFP '13, ACM, 2013, pp. 133–144.
[31] N. Swamy, C. Hriţcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss, J.-K. Zinzindohoue, S. Zanella-Béguelin, Dependent types and multi-monadic effects in F*, in: POPL '16, ACM, 2016, pp. 256–270.
[32] A. Nanevski, G. Morrisett, L. Birkedal, Hoare type theory, polymorphism and separation, J. Funct. Prog 18 (5-6) (2008) 865–911.
[33] P. W. O'Hearn, J. C. Reynolds, H. Yang, Local reasoning about programs that alter data structures, in: CSL '01, Vol. 2142 of LNCS, Springer, 2001, pp. 1–19.
[34] J. C. Reynolds, Separation logic: A logic for shared mutable data structures, in: LICS '02, 2002, pp. 55–74.
[35] A. Chlipala, G. Malecha, G. Morrisett, A. Shinnar, R. Wisnesky, Effective interactive proofs for higher-order imperative programs, in: ICFP '09, ACM, 2009, pp. 79–90.
[36] A. Charguéraud, F. Pottier, Verifying the correctness and amortized complexity of a union–find implementation in separation logic with time credits, Journal of Automated Reasoning 62 (3) (2019) 331–365.
[37] R. Ley-Wild, A. Nanevski, Subjective auxiliary state for coarse-grained concurrency, in: POPL '13, ACM, 2013, pp. 561–574.
[38] E. Mullen, S. Pernsteiner, J. R. Wilcox, Z. Tatlock, D. Grossman, œuf: Minimizing the Coq extraction TCB, in: CPP '18, ACM, 2018, pp. 172–185.
[39] R. Kumar, E. Mullen, Z. Tatlock, M. O. Myreen, Software verification with ITPs should use binary code extraction to reduce the TCB, in: ITP '18, Vol. 10895 of LNCS, Springer, 2018, pp. 362–369.
[40] A. Anand, S. Boulier, C. Cohen, M. Sozeau, N. Tabareau, Towards certified meta-programming with typed Template-Coq, in: ITP '18, Vol. 10895 of LNCS, Springer, 2018, pp. 20–39.
[41] S. Katsumata, Relating computational effects by ⊤⊤-lifting, Information and Computation 222 (2013) 228–246.
[42] M. Dénès, coq/coq#4749: `Extract Constant` has no effect on functors, https://github.com/coq/coq/issues/4749 (2016).
[43] G. Gonthier, math-comp/math-comp#294: Dependent positive finfun, https://github.com/math-comp/math-comp/pull/294 (2019).
[44] R. Nieuwenhuis, A. Oliveras, Proof-producing congruence closure, in: RTA '05, Vol. 3467 of LNCS, Springer, 2005, pp. 453–468.
[45] W. Swierstra, A Hoare logic for the state monad, in: TPHOLs '09, Vol. 5674 of LNCS, Springer, 2009, pp. 440–451.

## Appendix A. The correctness of optimization rules of Section 5.1

Here we sketch a correctness proofs of the optimization rules of Section 5.1. In this appendix, $\mathsf{FV}(t)$ denotes the set of free variables in $t$.

**Theorem 2** (Correctness of the generalized $\iota$-reduction)**.** For any MiniML terms $t$ and $t'$ which verify $t \rightsquigarrow_\iota t'$, $t$ is contextually equivalent to $t'$.

*Proof.* Let us prove

$$t \rightsquigarrow_\iota^{\overline{cl}} t' \Rightarrow (\mathsf{match}\ t\ \mathsf{with}\ cl) =_{\mathsf{ctx}} t'$$

by induction on the derivation of $t \rightsquigarrow_\iota^{\overline{cl}} t'$ where $\overline{cl} = C_1(\overline{x_1}) \to u_1 \mid \cdots \mid C_n(\overline{x_n}) \to u_n$.

**Case 1:** $C_i(t_1, \ldots, t_m) \rightsquigarrow_\iota^{\overline{cl}} \mathsf{let}\ \overline{x_i} := t_1, \ldots, t_m\ \mathsf{in}\ u_i$
LHS and RHS are convertible.

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

$$\begin{aligned}
(\text{LHS}) &= \mathsf{match}\ (\mathsf{let}\ x := u\ \mathsf{in}\ t)\ \mathsf{with}\ \overline{cl} \\
&= \mathsf{let}\ x := u\ \mathsf{in}\ \mathsf{match}\ t\ \mathsf{with}\ \overline{cl} && (x \notin \mathsf{FV}(\overline{cl})) \\
&=_{\mathsf{ctx}} \mathsf{let}\ x := u\ \mathsf{in}\ t' && (\text{I.H.}) \\
&= (\text{RHS})
\end{aligned}$$

**Case 3:** $t_1 \rightsquigarrow_\iota^{\overline{cl}} t'_1 \wedge \cdots \wedge t_m \rightsquigarrow_\iota^{\overline{cl}} t'_m \Rightarrow$ $\begin{array}{l}\mathsf{match}\ t\ \mathsf{with} \\ \mid D_1(\overline{x_1}) \to t_1 \\ \mid \ldots \\ \mid D_m(\overline{x_m}) \to t_m\end{array}$ $\rightsquigarrow_\iota^{\overline{cl}}$ $\begin{array}{l}\mathsf{match}\ t\ \mathsf{with} \\ \mid D_1(\overline{x_1}) \to t'_1 \\ \mid \ldots \\ \mid D_m(\overline{x_m}) \to t'_m\end{array}$

If $t$ has a HNF (head normal form) $D_i(\overline{u})$ in the given context,

$$\begin{aligned}
(\text{LHS}) &= \mathsf{match}\ (\mathsf{match}\ t\ \mathsf{with} \\
&\qquad \mid D_1(\overline{x_1}) \to t_1 \\
&\qquad \mid \ldots \\
&\qquad \mid D_m(\overline{x_1}) \to t_m)\ \mathsf{with}\ \overline{cl} \\
&=_{\mathsf{ctx}} \mathsf{match}\ (\mathsf{let}\ \overline{x_i} := \overline{u}\ \mathsf{in}\ t_i)\ \mathsf{with}\ \overline{cl} \\
&= \mathsf{let}\ \overline{x_i} := \overline{u}\ \mathsf{in}\ \mathsf{match}\ t_i\ \mathsf{with}\ \overline{cl} && (x \notin \mathsf{FV}(\overline{cl}) \text{ for any } x \in \overline{x_i}) \\
&=_{\mathsf{ctx}} \mathsf{let}\ \overline{x_i} := \overline{u}\ \mathsf{in}\ t'_i && (\text{I.H.}) \\
&=_{\mathsf{ctx}} \mathsf{match}\ t\ \mathsf{with} \\
&\qquad \mid D_1(\overline{x_1}) \to t'_1 \\
&\qquad \mid \ldots \\
&\qquad \mid D_m(\overline{x_1}) \to t'_m \\
&= (\text{RHS})
\end{aligned}$$

Otherwise ($t$ has other HNF or has no HNF), no information can be obtained from LHS and RHS; thus, they are contextually equivalent.

□

In Rules (6), (7), (8), (11), and (12), LHSs and RHSs are convertible respectively. Those convertibilities of Rules (6), (7), and (12) can be easily checked by reduction. Thus, here we check Rules (8) and (11).

*Proof.* LHS and RHS in Rule (8) are convertible as follows.

$$\begin{aligned}
(\text{RHS}) &= \mathsf{let}\ x := t_1\ \mathsf{in}\ t_2\, u_1 \dots u_n \\
&= (t_2\, u_1 \dots u_n)[x := t_1] \\
&= t_2[x := t_1]\, u_1 \dots u_n && (x \notin \mathsf{FV}(\mathsf{u_1}, \dots, \mathsf{u_n})) \\
&= (\mathsf{let}\ x := t_1\ \mathsf{in}\ t_2)\, u_1 \dots u_n \\
&= (\text{LHS})
\end{aligned}$$

LHS and RHS in Rule (11) are convertible as follows.

$$\begin{aligned}
(\text{RHS}) &= \mathsf{let}\ y := t_1\ \mathsf{in}\ \mathsf{let}\ x := t_2\ \mathsf{in}\ t_3 \\
&= t_3[x := t_2][y := t_1] \\
&= t_3[y := t_1][x := t_2[y := t_1]] && (\text{substitution lemma}) \\
&= t_3[x := t_2[y := t_1]] && (y \notin \mathsf{FV}(t_3)) \\
&= \mathsf{let}\ x := (\mathsf{let} y := t_1\ \mathsf{in}\ t_2)\ \mathsf{in}\ t_3 \\
&= (\text{LHS})
\end{aligned}$$

□

**Theorem 3** (Correctness of Rule (9))**.** LHS and RHS in Rule (9) are contextually equivalent.

*Proof.* If $t$ has a HNF $C_i(\overline{u})$ in the given context,

$$\begin{aligned}
(\text{LHS}) &= (\mathsf{match}\ t\ \mathsf{with} \\
&\quad \mid C_1(\overline{x_1}) \to t_1 \\
&\quad \mid \dots \\
&\quad \mid C_n(\overline{x_n}) \to t_n \\
&\quad )\, u_1 \dots u_m \\
&=_{\mathsf{ctx}} t_1[\overline{x_i} := \overline{u}]\, u_1 \dots u_m \\
&= (t_1\, u_1 \dots u_m)[\overline{x_i} := \overline{u}] && (x \notin \mathsf{FV}(u_1, \dots, u_m) \text{ for any } x \in \overline{x_i}) \\
&=_{\mathsf{ctx}} \mathsf{match}\ t\ \mathsf{with} \\
&\quad \mid C_1(\overline{x_1}) \to t_1\, u_1 \dots u_m \\
&\quad \mid \dots \\
&\quad \mid C_n(\overline{x_n}) \to t_n\, u_1 \dots u_m \\
&= (\text{RHS})
\end{aligned}$$

Otherwise, no information can be obtained from LHS and RHS. □

**Theorem 4** (Correctness of Rule (10))**.** LHS and RHS in Rule (10) are contextually equivalent.

*Proof.* If $t$ has a HNF $C_i(\overline{u})$ in the given context,

$$\begin{aligned}
(\text{LHS}) &= \mathsf{match}\ t\ \mathsf{with} \\
&\qquad \mid C_1(\overline{x_1}) \to \lambda y_1 \ldots y_m.\, t_1 \\
&\qquad \mid \ldots \\
&\qquad \mid C_n(\overline{x_n}) \to \lambda y_1 \ldots y_m.\, t_n \\
&=_{\mathsf{ctx}} (\lambda y_1 \ldots y_m.\, t_i)[\overline{x_i} := \overline{u}] \\
&= \lambda y_1 \ldots y_m.\, t_i[\overline{x_i} := \overline{u}] \\
&=_{\mathsf{ctx}} \lambda y_1 \ldots y_m.\, \mathsf{match}\ t\ \mathsf{with} \\
&\qquad\qquad\qquad \mid C_1(\overline{x_1}) \to t_1 \\
&\qquad\qquad\qquad \mid \ldots \\
&\qquad\qquad\qquad \mid C_n(\overline{x_n}) \to t_n \\
&= (\text{RHS})
\end{aligned}$$

Otherwise, no information can be obtained from LHS and RHS.[9] □

[9] Note that binders $\lambda y_1 \ldots y_m$ do not involve $t$.