# Specialization of Generic Array Accesses After Inlining (System Description)

Ryohei Tokuda Eijiro Sumii Akinori Abe
Graduate School of Information Sciences, Tohoku University, Sendai, Japan
tokuda@sf.ecei.tohoku.ac.jp sumii@ecei.tohoku.ac.jp abe@sf.ecei.tohoku.ac.jp

We have implemented an optimization that specializes type-generic array accesses after inlining of polymorphic functions in the native-code OCaml compiler. Polymorphic array operations (read and write) in OCaml require runtime type dispatch because of ad hoc memory representations of integer and float arrays. It cannot be removed even after being monomorphized by inlining because the intermediate language is mostly untyped. We therefore extended it with explicit type application like System F (while keeping implicit type abstraction by means of unique identifiers for type variables). Our optimization has achieved up to 21% speed-up of numerical programs.

## 1 Introduction

### 1.1 Background

Representation of primitive values such as floating-point numbers is a classical problem in the implementation of polymorphic languages since ad hoc representations hinder uniform treatment of values. The classical way to overcome this difficulty is to fit every value in one machine word by heap-allocating multi-word data (called *boxing*) and manipulating them through pointers. Although simple, this method is inefficient especially for numerical computations because of the heap allocation and pointer dereferences. The cost of garbage collection due to the frequent allocations is particularly problematic.

More sophisticated implementation methods for polymorphism have also been devised. Leroy [8] and Shao [10] adopt a mixture of specialized and uniform representations, where the cost of conversions between different representations (such as flat arrays vs. arrays of boxed values) is non-trivial. Another method is to pass the types at runtime [7, 9, 14], where the construction of the type representations itself incurs an overhead.

OCaml takes an ad hoc approach to the problem: it mainly adopts uniform representation but uses unboxed representations for "local" floating-point numbers (i.e., ones that do not escape a function's body) and, in particular, for *arrays* of floating-point numbers (as well as records with fields of floating-point numbers only). Such specialized representation of `float array` means that polymorphic array accesses (such as setting and getting the elements) need to *dynamically* check the type of the array and make a case branch when it is a `float array`. These dynamic checks generally incur runtime overheads, which the standard OCaml compiler tries to remove if the monomorphic type of the array is statically known.[1]

[1] Standard ML (as well as OCaml's `bigarray`) takes another ad hoc approach: its Basis Library offers a monomorphic module `RealArray` (along with other monomorphic array modules of a common interface) for an unboxed representation. There is also a similar proposals [6, 15] for `array` in OCaml.

Jeremy Yallop and Damien Doligez (Eds.)
ML/OCaml 2015
EPTCS 241, 2017, pp. 45–53, doi:10.4204/EPTCS.241.4

## 1.2 Problem

Despite the aforementioned specialization of generic array accesses, the standard OCaml compiler fails to apply the specialization when the monomorphic type is known *after* inlining of functions, because of a lack of type information in the intermediate language.

To see the problem concretely, consider the following program:

```
let get0 a = a.(0)  (* 'a array -> 'a *)
let i = get0 int_array
let f = get0 float_array
```

Even if the polymorphic function `get0` is inlined, the array accesses `int_array.(0)` and `float_array.(0)` are *not* specialized and are considered generic, incurring the runtime overheads of case branches over the dynamic types of the (obviously monomorphic) arrays.

This problem is due to the internal representation of the partial type information attached (only) to array accesses in the intermediate language, which is defined in the OCaml compiler as (roughly speaking):

```
type array_kind =
  | Pgenarray   (* generic           *)
  | Pintarray   (* int               *)
  | Pfloatarray (* float             *)
  | Paddrarray  (* address (pointer) *)
```

The `array_kind` "`Pgenarray`" means runtime dispatch over the dynamic type of the elements of an array.

For example, the program above can be annotated with this (partial) type information like

```
let get0 a = a.{Pgenarray}(0)
let i = get0 int_array
let f = get0 float_array
```

where `{}` denotes the internal type annotation with `array_kind`. Obviously, just inlining the function `get0` does not specialize the generic read operations:

```
let i = int_array.{Pgenarray}(0)
let f = float_array.{Pgenarray}(0)
```

## 1.3 Key Ideas

Our idea is to add explicit type information like System F to the mostly untyped intermediate language `lambda` of OCaml. That is, we basically extend the intermediate language with type abstractions and applications.

For instance, the example above can be type-annotated like:

```
let get0{'a} a = a.{'a}(0)
let i = get0{Pintarray} int_array
let f = get0{Pfloatarray} float_array
```

We add the formal type parameter `{'a}` in the definition of `get0`. Then, we attach the type information `{'a}` on the read access `a.(0)` to the array `a` of polymorphic type `'a array`. We then explicitly denote type applications by annotating `get0` with `{Pintarray}` and `{Pfloatarray}`. The generic array accesses can now be specialized by inlining as:

```
let i = int_array.{Pintarray}(0)
let f = float_array.{Pfloatarray}(0)
```

For type application, we indeed extended the intermediate language. For abstraction, we actually used globally unique identifiers for type variables to avoid introducing a new binder, as OCaml does for typing; in exchange, we have to specify *which* type variable to instantiate at the application side.

For example, the foregoing program is now represented like:

```
let get0 (* {'a} *) a = a.{'a}(0)
let i = get0{'a↦I} int_array
let f = get0{'a↦F} float_array
```

In the definition of `get0`, we omit the type abstraction `(* {'a} *)` while annotating the read access with the implicitly bound type variable `'a`. We then annotate the applications of `get0` with a mapping from the type variable `'a` to an `array_kind`, like `{'a↦I}` and `{'a↦F}`. When inlining `get0`, we replace the occurrence of `'a` according to the given mapping, resulting in

```
let i = int_array.{I}(0)
let f = float_array.{F}(0)
```

as desired.

Of course, not all polymorphic functions can be inlined completely (because of code size, for example). In such cases, some type variables still remain and are compiled as generic array accesses with dynamic type dispatch.

## 2 Implementation

In this section, we describe details of our implementation based on the native-code OCaml compiler. We have made the following changes to the intermediate language `lambda` (resp. `clambda`) before (resp. after) closure conversion.

### 2.1 Refining the type information

First, we replace `Pgenarray` (type of generic arrays) with `Ptvar of int` (type variable with an integer identifier) in `array_kind` to specify *which* type variable the generic type refers to (like `{'a}` in `a.{'a}(0)` instead of `Pgenarray` in `a.{Pgenarray}(0)`) as follows[2]:

```
type array_kind =
  Ptvar of int (* id *) | Pintarray | Pfloatarray | Paddrarray
```

### 2.2 Making type applications explicit

Second, we add `Lspecialized` to the intermediate languages for explicitly representing type applications.

[2] Precisely speaking, we still keep `Pgenarray` for places where our current implementation abandons specialization, such as functors and GADTs.

```
type lambda = (* ditto for clambda *)
| ...(* same as before *)...
| Lspecialized of lambda * kind_map (* type application *)
and kind_map = (int * array_kind) list (* association list *)
```

The constructor `Lspecialized` has two parameters: the first parameter (of type `lambda`) is a polymorphic function to be specialized, while the second (type `kind_map`) is a type mapping described above, such as {'a↦I} and {'a↦F}. We insert `Lspecialized` to every occurrence of a let-bound polymorphic variable during the translation from `typedtree` (typed AST) to `lambda`.[3] More specifically, for every variable occurrence, we compare the monomorphic type of the variable (annotated in the AST) with the polymorphic type stored in the type environment and, if the latter type is indeed polymorphic, insert `Lspecialized` with `kind_map` recovered from the comparison by means of a one-directional unification. (In principle, this mapping is already known during type inference but is discarded in the current OCaml compiler. We avoided modifying the type inference because of its complexity.)

## 2.3 Recording and following the renaming of type variables

Finally, we need to record and follow renaming of type variables exported via `.cmx` files to prevent inconsistencies with `.cmi` files.

Suppose, for example, that we compile a source file `a.ml` including function `get0`

a.ml

```
let get0 a = a.{'a}(0)
```

with the interface file:

a.mli

```
val get0 : 'a array -> 'a
```

On one hand, the `.mli` is compiled into a `.cmi` file:

a.cmi

```
(* pseudo-code for the binary *)
val get0 : 'a array -> 'a
```

On the other hand, the implementation `a.ml` is compiled separately from the interface `a.mli`. Even the types are inferred independently—they are only *checked* against the compiled interface `a.cmi` *after* inference. As a result, the type variable `'a` may be given a completely different identifier in the AST `a.cmx` generated for inlining:

a.cmx

```
(* pseudo-code for the AST *)
get0 a = a.{'b}(0)
```

[3] Thus, the only possible first parameter for `Lspecialized` is actually an occurrence of a local (`Lvar`) or global (module access) variable, so it is also possible to attach the `kind_map` to those variable occurrences of instead of introducing `Lspecialized`. We did not take this approach because modifying the module access primitive (`Pgetglobal`) seemed more complicated than simply adding `Lspecialized`.

This inconsistency is problematic for our scheme, where `get0` is applied like `A.get0{'a↦I}int_array` according to its type in the interface `a.cmi`. We fixed this by adjusting the type variable identifiers in a `.cmx` according to the corresponding `.cmi` just before generating the former:

`a.cmx` (adjusted)

```
(* pseudo-code for the AST *)
get0 a = a.{'a}(0)
```

Moreover, suppose that the function `A.get0` is used from another file `b.ml`. When the interface `a.cmi` is imported, the type variable `'a` is renamed for the sake of uniqueness inside the importing module `B`

`b.ml`

```
open A
(* val get0 : 'c array -> 'c *)

let i = get0{'c↦I} int_array (* an example using A.get0 *)
```

and becomes inconsistent with the implementation `a.cmx`. We fixed this inconsistency by remembering the renaming such as `'a↦'c` in a global table at import time, and applying it before inlining function bodies such as `a.{'a}(0)`.

`b.ml` (after inlining)

```
open A
(* renaming table for A is 'a↦'c *)
(* val get0 : 'c array -> 'c *)

let i = int_array.{('c↦I)('a↦'c)'a}(0) (* correctly inlined *)
```

## 3 Experiments

We have implemented the above specialization[4] on top of the 4.02 branch of OCaml as of May 6, 2015, and measured its effects for the numerical programs in Table 1. "Simple" is a program that adds all elements of an array, where all the accesses are made through polymorphic functions. "Random" makes generic array accesses, randomly switching between `int array` and `float array` (in addition, the pseudo-random number generator internally makes monomorphic array accesses). The other benchmarks are realistic, naturally written numerical programs: "DKA" stands for Durand-Kerner-Aberth (a method for finding a root of a complex polynomial), "FFT" is a fast Fourier transform, "K-means" is a clustering method used for data mining and machine learning, "LD" stands for the Levinson-Durbin recursion for time series analysis, "LU" is the LU decomposition in linear algebra, "NN" is a neural network program, and "QR" is the QR decomposition (again in linear algebra). We have hand-tuned the timing of garbage collections. We compiled all the files (including the standard library modules) with `ocamlopt -inline 10000000` (and `-unsafe` for the benchmark programs[5]).

The results (on Ubuntu Linux 14.04, Intel(R) Core(TM) i7-2677M 1.80 GHz—fixed at 1.0 GHz by `cpufreq` to avoid experimental errors caused by Intel Turbo Boost, which is sensitive to changes in temperature!—and 4 GiB DDR3 SDRAM) are also in Table 1 and can be explained as follows:

[4] Our compiler is available from: `https://github.com/nomaddo/ocaml`

[5] Their source code is available from: `https://github.com/nomaddo/ocaml-numerical-analysis/tree/bench`

| Program | All | Gen-before | | Gen-after | |
|---|---|---|---|---|---|
| Simple | 219,999,999 | 219,999,999 | (100.0%) | 0 | (0.0%) |
| Random | 1,200,000,001 | 599,999,999 | (50.0%) | 0 | (0.0%) |
| DKA | 38,436,359 | 12,899,744 | (33.6%) | 1,748 | (0.0%) |
| FFT | 222,298,106 | 6,291,450 | (2.8%) | 0 | (0.0%) |
| K-means | 16,996,284 | 13,022,928 | (76.6%) | 11,562,804 | (68.0%) |
| LD | 1,583,266,345 | 250,025,000 | (15.8%) | 0 | (0.0%) |
| LU | 555,548,601 | 7,603,452 | (1.4%) | 7,589,706 | (1.4%) |
| NN | 1,324,325,917 | 671,112,411 | (50.7%) | 368,199,383 | (27.8%) |
| QR | 577,566,417 | 571,855,214 | (99.0%) | 133,758 | (0.0%) |

| Program | Time-before | Time-after | Speed-up |
|---|---|---|---|
| Simple | 0.372 s | 0.354 s | 5% |
| Random | 4.619 s | 3.939 s | 15% |
| DKA | 0.370 s | 0.363 s | 2% |
| FFT | 6.119 s | 6.136 s | 0% |
| K-means | 0.406 s | 0.401 s | 1% |
| LD | 7.324 s | 7.115 s | 3% |
| LU | 3.261 s | 3.263 s | 0% |
| NN | 12.20 s | 11.75 s | 4% |
| QR | 6.255 s | 4.931 s | 21% |

Table 1: Results of experiments: "All" is the number of all array accesses, and "Gen-before" (resp. "Gen-after") is generic array accesses before (resp. after) the specialization, and "Speed-up" is $(\text{Time-after} - \text{Time-before})/\text{Time-before} \times 100\%$.

- "Simple", "DKA", and "LD" show modest speed-up because all the generic accesses are specialized while most of the execution time is still spent on floating-point operations.
- "Random" exhibits considerable improvement since the generic accesses are removed and the program does not perform any other significant computation.
- "FFT" and "LU" contain only a relatively small number of generic accesses in the first place because of the monomorphic coding style.
- "K-means", "NN", and "LU" include polymorphic array access functions that are not inlined at all, probably because of closure sharing of the OCaml compiler. For instance, the following function `foldi`

```
let foldi f init x =
  snd (Array.fold_left
         (fun (i, acc) xi -> (i+1, f i acc xi))
         (0, init)
         x)
```

  is never inlined, since an argument `f` of `foldi` appears free in the anonymous function `fun (i, acc) xi -> (i+1, f i acc xi)`.
- The high-level coding style of QR almost exclusively uses polymorphic functions such as `Array.map`

and `Array.fold_left` as opposed to low-level index accesses, and achieves 21% speed-up thanks to the almost complete specialization.

## 4 Related Work

While we have extended the partial explicit type information for array access operations in the intermediate language(s) of OCaml, more explicitly typed intermediate languages have already been studied extensively:

- Harper and Morrisett [7] formalized a translation from the implicitly typed ML core language to an explicitly typed intermediate language $\lambda_i^{ML}$, which is a variant of $F_\omega$ extended with intensional type analysis. Crary and Weirich [4] further extended $\lambda_i^{ML}$, generalizing the type language.
- TIL [14], a Standard ML compiler with intensional polymorphism, adopted $\lambda_i^{ML}$ as the intermediate language for type-directed optimizations and applied conventional optimizations (inlining, uncurrying, common sub-expression elimination, etc) to the type-level language to reduce its overheads, in particular the construction of type representations.
- FLINT [11, 12], the intermediate language of SML/NJ [1], used directed acyclic graphs instead of trees as type representations for scalability against large types.
- GHC uses System $F_C$[13], an extension of $F_\omega$ with type coercion for uniformly supporting a wide variety of features such as GADTs [16] and associated types [3, 2].

Compared with these fully typed intermediate languages, our extension (`Ptvar` and `Lspecialized`) to the partial type information (`array_kind`) for array access operations in OCaml is ad hoc but small and relatively easy, with less than 1000 lines of modification to (hundreds of thousands lines of) the native-code compiler.

## 5 Conclusion

We have made a relatively simple modification to the native-code OCaml compiler to specialize generic array accesses after inlining, and observed modest or significant speed-ups for numerical programs.

Currently, a new intermediate language `flambda` [5] is under development (independently of our work) in the `trunk` branch of OCaml. We expect that it solves problems of the current `lambda` (like closure sharing hinders inlining as observed in Section 3) as well as making our approach even more effective by enabling the specialization of recursive functions (which are never inlined by the current OCaml compiler), for example.

Although our experiments focused on the efficiency of floating-point programs, the optimization may also be effective for generic functions (such as `Array.map` and `Array.fold_left`) applied to integer or pointer arrays. It would also be interesting future work to adapt an approach similar to ours for specializing other operations than array accesses, such as polymorphic comparisons and unboxing local variables.

## Acknowledgments

We thank Jacques Garrigue for his help on hacking the OCaml compiler and the anonymous reviewers for valuable comments. This work was partially supported by JSPS KAKENHI Grant Numbers JP22300005,

JP25540001, JP15H02681, JP16K12409, and by Mitsubishi Foundation Research Grants in the Natural Sciences.

## References

[1] *Standard ML of New Jersey*. http://www.smlnj.org.

[2] Manuel M. T. Chakravarty, Gabriele Keller & Simon L. Peyton Jones (2005): *Associated type synonyms*. In: *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming, ICFP 2005, Tallinn, Estonia, September 26-28, 2005*, pp. 241–253, doi:10.1145/1086365.1086397.

[3] Manuel M. T. Chakravarty, Gabriele Keller, Simon L. Peyton Jones & Simon Marlow (2005): *Associated types with class*. In: *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, pp. 1–13, doi:10.1145/1040305.1040306.

[4] Karl Crary & Stephanie Weirich (1999): *Flexible Type Analysis*. In: *Proceedings of the fourth ACM SIGPLAN International Conference on Functional Programming (ICFP '99), Paris, France, September 27-29, 1999.*, pp. 233–248, doi:10.1145/317636.317906.

[5] *Optimisation with Flambda*. http://caml.inria.fr/pub/docs/manual-ocaml/flambda.html.

[6] Alan Frisch (2015): *About unboxed float arrays*. https://www.lexifi.com/blog/about-unboxed-float-arrays.

[7] Robert Harper & J. Gregory Morrisett (1995): *Compiling Polymorphism Using Intensional Type Analysis*. In: *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*, pp. 130–141, doi:10.1145/199448.199475.

[8] Xavier Leroy (1992): *Unboxed Objects and Polymorphic Typing*. In: *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 177–188, doi:10.1145/143165.143205.

[9] Ronald Morrison, Alan Dearle, Richard C. H. Connor & Alfred L. Brown (1991): *An Ad Hoc Approach to the Implementation of Polymorphism*. *ACM Trans. Program. Lang. Syst.* 13(3), pp. 342–371, doi:10.1145/117009.117017.

[10] Zhong Shao (1997): *Flexible Representation Analysis*. In: *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming (ICFP '97), Amsterdam, The Netherlands, June 9-11, 1997.*, pp. 85–98, doi:10.1145/258948.258958.

[11] Zhong Shao (2000): *Typed common intermediate format*. *ACM SIGSOFT Software Engineering Notes* 25(1), p. 82, doi:10.1145/340855.341019.

[12] Zhong Shao, Christopher League & Stefan Monnier (1998): *Implementing Typed Intermediate Languages*. In: *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP '98), Baltimore, Maryland, USA, September 27-29, 1998.*, pp. 313–323, doi:10.1145/289423.289460.

[13] Martin Sulzmann, Manuel M. T. Chakravarty, Simon L. Peyton Jones & Kevin Donnelly (2007): *System F with type equality coercions*. In: *Proceedings of TLDI'07: 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Nice, France, January 16, 2007*, pp. 53–66, doi:10.1145/1190315.1190324.

[14] David Tarditi, J. Gregory Morrisett, Perry Cheng, Christopher A. Stone, Robert Harper & Peter Lee (1996): *TIL: a type-directed, optimizing compiler for ML (with retrospective)*. In: *20 Years of the ACM SIGPLAN Conference on Programming Language Design and Implementation 1979-1999, A Selection*, pp. 554–567, doi:10.1145/989393.989449.

[15] Leo White (2015): *Remove float array optimisation*. https://github.com/ocaml/ocaml/pull/163.

[16] Hongwei Xi, Chiyan Chen & Gang Chen (2003): *Guarded recursive datatype constructors.* In: *Conference Record of POPL 2003: The 30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New Orleans, Louisisana, USA, January 15-17, 2003*, pp. 224–235, doi:10.1145/640128.604150.