Apple Array Allocation

V. E. McHale
Northern Trust
Chicago, IL, USA
vamchale@gmail.com

Abstract

Array languages like J and APL suffer from a lack of embeddability in implementations. Here we present statically determined memory allocation for flat, immutable arrays used in the Apple array system, a JIT compiler for an expression-oriented functional language. The method is a straightforward extension of liveness analysis familiar to compiler writers. Ownership is simple and Apple does not constrain memory management in the host language. Two implementations—one in Python and one in R—are exhibited.

1 Introduction

1.1 Background

Array languages such as APL and J suffer from undeserved obscurity [3]. In order to make them more useful to enthusiasts, one might hope to embed them, thus providing access to developed libraries, e.g. matplotlib [5].

APL implementations typically use reference-counting [4, p. 47] for performance reasons, but this imposes on the calling code—one must increment the reference count and delegate each array to a garbage collector. Our arrays are pointers managed by the system's `malloc` and `free`. By providing a primitive interface we grant the host language a more equal status; this is arguably how C has succeeded [6] and how APL on .NET failed [4, p. 12].

1.2 Apple Array System

The Apple JIT compiler is a function

```
apple_compile :: ( FunPtr (CSize -> Ptr a)
                 , FunPtr (Ptr a -> ())
                 )
              -> String -> Bytes
```

taking a pointer to `malloc`, a pointer to `free`, and source code as input and returning machine code; the compiler only generates these two foreign function calls and there is no runtime system.

This simplifies linking in the JIT; the lack of a garbage collector makes for a good example exploring compiler architectures—one could imagine compiling on one computer and then sending machine code elsewhere for execution.

Conference'17, July 2017, Washington, DC, USA
2024. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1.2.1 Language. The language is expression-oriented and all data are immutable. Recursion is not supported; all looping is implicit. This is hardly burdensome for array programmers, who favor vector solutions already [8, Chapter. 31].

The below computes the Kullback-Liebler divergence using a fold and a zip:

```
\p.\q. (+)/([x*_.(x%y)]`p q)
```

In general, the language has the scope of a calculator, plus select array operations.

1.2.2 Flat Array Types. Apple arrays are completely flat in memory, consisting of rank, dimensions, and data laid out contiguously.

A `2x4` array:

0	1	2	3	4	5	6	7	8	9	10
2	2	4	a_{00}	a_{01}	a_{02}	a_{03}	a_{10}	a_{11}	a_{12}	a_{13}

All elements are flat; there are only two primitive types: 64-bit integers and 64-bit floats. Arrays of tuples are supported but not arrays of tuples of arrays; there are no user-defined types. Arrays of arrays are treated as higher-dimensional arrays. These limitations are typical of array languages and are enough for nontrivial computation, for instance neural networks.

Such constraints, with extra bookkeeping information added during IR generation, are responsible for the surprising fact that memory allocation can be completely determined at compile time.

2 Method

2.1 Compiler Pipeline

Our work is based on established liveness algorithms; we define `uses` and `defs` on statements (in terms of arrays), compute liveness [2, pp. 213-216], and then construct live intervals [7].

$$\begin{tikzcd}
\cdots \arrow[d, "\text{IR Generation}"] \\
\boxed{\texttt{[Stmt]}} \arrow[d, "\text{Liveness Analysis}"] \\
\boxed{\texttt{[(Stmt, Liveness)]}} \arrow[d, "\text{Insert Frees}"] \\
\boxed{\texttt{[Stmt]}}
\end{tikzcd}$$

The IR used in the Apple compiler is sequences of statements and expressions, viz.

```
data Exp = Const Int
         | Reg Temp
         | At ArrayPtr
         ...

data Stmt = MovTemp Temp Exp
          | Write ArrayPtr Exp
          -- label, register, size
          | Malloc Lbl Temp Exp
          | CondJump Exp Loc
          | Jump Loc | Label Loc
          | Free Temp
          ...
```

Expressions can read from memory via `At` and `Stmts` make use of memory access for writes, allocations etc. A function

```
aeval :: Expr -> IRM (Temp, Lbl, [Stmt])
```

translates an array expression, assigning a `Lbl` for subsequent access and associating the labeled array with a `Temp` to be used to free it.

All accesses to an array use an `ArrayPtr`, which specifies the `Lbl` for tracking within the compiler. In this way we avoid being confounded by multiple pointers to the same array.

```
-- register, offset, label
data ArrayPtr = ArrayPtr Temp (Maybe Exp) Lbl
```

Recall that references are not supported, so it is impossible for an array access to keep another array live.

Then we have:

```
defs :: Stmt -> Set Lbl
defs (Malloc l _ e) = singleton l
defs _              = empty

uses :: Stmt -> Set Lbl
uses (Write (ArrayPtr _ _ l) e) =
    insert l (usesE e)
uses ...
```

By tracking arrays via these labels, one can access the array from different `Temps`; this is necessary for generating efficient code.

After liveness analysis, we have liveness information for each `Stmt`, viz.

```
data Liveness =
  Liveness { ins :: Set Lbl, outs :: Set Lbl }
```

With this we compute live intervals for each `Lbl`, as in the literature [7].

We then insert a `free` when the live interval for an array ends by looking up the `Temp` associated with the `Lbl`.

2.2 Generation

We must take care when arranging loops; the loop should exit by continuing rather than jumping to an exit location. This ensures that the `frees` inserted at the end of live intervals are always reachable.

For instance, we do not write:

```
apple_0:
(condjump (< (reg r_2) (int 2)) apple_1)
(movtemp r_0
  @(ptr r_1+(+ (asl (reg r_2) (int 3)) (int 16))))
(movtemp r_2 (+ (reg r_2) (int 1)))
(jump apple_0)

apple_1:
```

But rather:

```
(condjump (≥ (reg r_2) (int 2)) apple_1)

apple_0:
(movtemp r_0
  @(ptr r_1+(+ (asl (reg r_2) (int 3)) (int 16))))
(movtemp r_2 (+ (reg r_2) (int 1)))
(condjump (< (reg r_2) (int 2) apple_0)

apple_1:
```

Had we written the former, the `free` would be unreachable, viz.

```
apple_0:
(condjump (< (reg r_2) (int 2)) apple_1)
(movtemp r_0
  @(ptr r_1+(+ (asl (reg r_2) (int 3)) (int 16))))
(movtemp r_2 (+ (reg r_2) (int 1)))
(jump apple_0)
(free r_2)

apple_1:
```

This is not the case with the latter:

```
(condjump (≥ (reg r_2) (int 2)) apple_1)

apple_0:
(movtemp r_0
  @(ptr r_1+(+ (asl (reg r_2) (int 3)) (int 16))))
(movtemp r_2 (+ (reg r_2) (int 1)))
(condjump (< (reg r_2) (int 2) apple_0)
(free r_2)

apple_1:
```

Since jumps are only generated by the compiler in a few cases, we can guarantee that generated code is correct by being careful.

3 Embeddings

The Apple JIT has been embedded in Python as an extension module [1]. Observe the following interaction:

```
>>> import apple
>>> area=apple.jit('''
λas.λbs.
    { Σ ⇐ [(+)/x]
    ; 0.5*abs.(Σ ((*)`as (1⊖bs))
        - Σ ((*)`(1⊖as) bs))
    }
''')
>>> import numpy as np
>>> apple.f(area,
    np.array([0,0,3.]),
    np.array([0,4,4.]))
6.0
```

There are wrappers for calling the JIT from R as well [9]:

```
> shoelace<-jit("
λas.λbs.
    { Σ ⇐ [(+)/x]
    ; 0.5*abs.(Σ ((*)`as (1⊖bs))
        - Σ ((*)`(1⊖as) bs))
    }
")
> run(shoelace,c(0,0,3),c(0,4,4))
[1] 6
```

Thus we can run Apple code on NumPy arrays and R vectors without frustrating the garbage collector.

We can also compile to an object file and then call from C, viz.

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

typedef double F;typedef int64_t I;
typedef void* U;

#define V(n,xs,p) \
  { \
    p=malloc(16+8*n); \
    I* i_p=p;*i_p=1;i_p[1]=n; \
    memcpy(p+16,xs,8*n); \
  }

extern F shoelace(U, U);

int main(int argc, char *argv[]) {
    F xs[] = {0,4,4};
    F ys[] = {0,0,3};
    U x; V(3,xs,x);
    U y; V(3,ys,y);
    printf("%f\n", shoelace(x,y));
    free(x);free(y);
}
```

Thus Apple code works with languages such as C with more primitive provisions.

4 Coda

Apple has the potential to be far more efficient; one could consolidate allocations, e.g.

```
 irange 0 20 1 ++ irange 0 10 1
```

performs one allocation for each `irange` but this could be consolidated into one—`irange 0 20 1` is inferred to have type `Vec 20 int` in the existing compiler; with liveness and size information one could allot arrays whose live intervals do not overlap to the same location in memory.

This constrained high-level language occupies a sweet spot in compilers: because we control IR generation, we can do analyses that would be thwarted by pointer aliasing in C, and because we do not support references we simplify tracking.

References

[1] 2024. Extending Python with C or ++. https://docs.python.org/3/extending/extending.html. Accessed: 2024-04-11.

[2] Andrew Appel. 1998. *Modern Compiler Implementation in ML.* Cambridge University Press.

[3] Aaron W. Hsu and Rodrigo Girão Serrão. 2023. U-Net CNN in APL: Exploring Zero-Framework, Zero-Library Machine Learning. In *Proceedings of the 9th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming* (Orlando, FL, USA) *(ARRAY 2023)*. Association for Computing Machinery, New York, NY, USA, 22–35. https://doi.org/10.1145/3589246.3595371

[4] Roger K. W. Hui and Morten J. Kromberg. 2020. APL since 1978. *Proc. ACM Program. Lang.* 4, HOPL, Article 69 (jun 2020), 108 pages. https://doi.org/10.1145/3386319

[5] J. D. Hunter. 2007. Matplotlib: A 2D graphics environment. *Computing in Science & Engineering* 9, 3 (2007), 90–95. https://doi.org/10.1109/MCSE.2007.55

[6] Stephen Kell. 2017. Some were meant for C: the endurance of an unmanageable language. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Vancouver, BC, Canada) *(Onward! 2017)*. Association for Computing Machinery, New York, NY, USA, 229–245. https://doi.org/10.1145/3133850.3133867

[7] Massimiliano Poletto and Vivek Sarkar. 1999. Linear scan register allocation. *ACM Trans. Program. Lang. Syst.* 21, 5 (sep 1999), 895–913. https://doi.org/10.1145/330249.330250

[8] Roger Stokes. 2015. *Learning J.*

[9] Hadley Wickham. [n. d.]. R's C interface. http://adv-r.had.co.nz/C-interface.html#calling-c. Accessed 2024-04-11.