Apple Array Allocation

ANONYMOUS AUTHOR(S)

Array languages like J and APL suffer from a lack of embedability in implementations. Adroit memory management can make embedding easier; one would like to avoid thinking about ownership across two garbage collectors. Here we present statically determined memory allocation for flat, immutable arrays used in the Apple array system, a compiler for an expression-oriented functional language. Arrays are usable as pointers, with no dependence on a garbage collector. Ownership is simple and Apple code does not constrain memory management in the host language. Two embeddings—one in Python and one in R—are exhibited.

1 INTRODUCTION

Array languages such as APL and J suffer from undeserved obscurity [2]. In order to make wider use of array languages, one might hope to embed them.

1.1 Background

Rather than comparing to competitors such as NumPy [1], we take inspiration from C—we would like an array language that does not impose on its host language [4]. C is almost vulgar in offering pointers as arrays; the typical APL implementation uses reference-counting [3, p. 47] for performance reasons. But this imposes on the calling code, requiring care to increment the reference count

By compiling high-level array expressions to a C procedure with all allocations and frees predetermined, we reduce the demands of calling Apple code in other environments.

	Input	Array	Compiles To
C	Sequence of steps	Pointer	Machine code in object file
J	Expression	Reference-counted	Action in environment

1.2 Apple Array System

The Apple compiler

1.2.1 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}

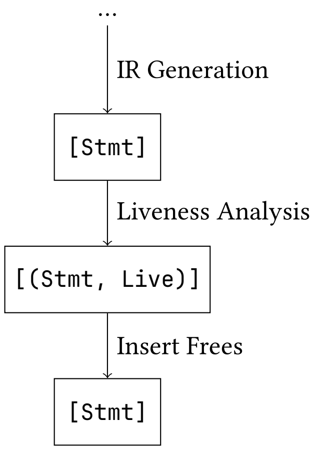
All elements are flat; there are only two primitive types: 64-bit integers and 64-bit floats. Arrays of tuples are supported but not arrays of tuples of arrays; there are no user-defined types.

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. Though such limited types may seem spartan to a functional programmer, this is enough to stand toe-to-toe with NumPy.

2 METHOD

2.1 Compiler Pipeline

Our work is based on established liveness algorithms; we annotate statements with `uses` and `defs` and thence compute live intervals [5] for arrays.



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
          | Malloc Lbl Temp Exp -- label, register, size
          | 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 the `ArrayPtr` type, which specifies the `Lbl` for tracking within the compiler.

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

Then one has

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

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

Note that with such labels one can access the array from different `Temps`; this is necessary for generating efficient code in the compiler.

With `defs` and `uses` thus defined, we compute live intervals for `Stmts` [5].

```
data Live = Live { done :: Set Lbl, new :: Set Lbl }
```

For each `(Stmt, Live)`, when we encounter a label in the `done` set, we look up the `Temp` to free it and insert a `free` statement.

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))))
(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))))
(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))))
(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))))
(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

Apple has been embedded in Python and R. Observe the following interactions:

```
>>> 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
```

```
> source("./apple.R")
> 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.

Apple code can be called from C on its own terms, unlike NumPy or J, viz.

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

#define R return
#define DO(i,n,a) {I i;for(i=0;i<n;i++){a;}}

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

typedef struct Af {I rnk; I* dim; F* xs;} Af;

U poke_af (Af x) {
    I rnk = x.rnk;
    I t = 1;
    DO(i,rnk,t*=x.dim[i]);
    U p = malloc(8+8*x.rnk+8*t);
    I* i_p = p;F* f_p = p;
    *i_p = rnk;
    DO(i,rnk,i_p[i+1]=x.dim[i]);
    DO(i,t,f_p[i+1+rnk]=x.xs[i]);
    R p;
}

extern F shoelace(U, U);

int main(int argc, char *argv[]) {
    F xs[] = {0,4,4};
    F ys[] = {0,0,3};
    I d[] = {3};
    Af a = {1,d,xs};
    Af b = {1,d,ys};
    U x = poke_af(a);U y = poke_af(b);
    printf("%f\n", shoelace(x,y));
    free(x);free(y);
}
```

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 do something like linear register allocation in which arrays are assigned to memory slots.

This vindicates flat arrays; while banning arrays of functions &c. may not be obvious to a functional programmer, studying this particular set of constraints has potential; one could use it to implement a compiler for a high-level language that produces performant and portable code.

REFERENCES

[1] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. 2020. Array programming with NumPy. *Nature* 585, 7825 (2020), 357–362. https://doi.org/10.1038/s41586-020-2649-2

[2] 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

[3] 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

[4] 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

[5] 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