No branching, no recursion, no problem

Lexical scoping with arrays

|  |  |
| --- | --- |
| Author Name  Author Affiliation *as many lines as needed press* Shift + Return *to insert line breaks*  Author email | Author Name  Author Affiliation *see Section 1 on how to add or delete authors*  Author email |

What would a compiler implementation look like if it did not use any conditionals, branching control flow, or recursion? This programming pearl describes a direct, straightforward method of lifting functions and resolving lexical scope using only array operations. The exposition focuses on the clarity of the presented methods and ideas. It gives some perspective on the construction of such passes and the array-oriented thinking used to arrive at a suitable implementation. This design is part of an effort to build a self-hosting, high-performance array language compiler, Co-dfns, that efficiently executes on modern architectures like the GPU.

# Introduction

Parallelism permeates modern computing architectures. Traditional algorithmic descriptions of compiler transformations ignore this. Furthermore, the near universal approach to thinking about compiler transformation centers around recursive algorithms with a high degree of branching control flow, in the form of dispatch over the types of the nodes in an AST. In contrast, architectures like GPUs, and, increasingly, multi-core CPUs, reward vectorizable code through large numbers of independent processing units, large vector lanes, and fast SIMD instruction handling. Attempts to implement compiler transformations on the GPU tend to focus strictly on performance, which obfuscates the core ideas behind the process and design of writing a compiler pass that exhibits favorable characteristics for parallel execution, namely, minimal thread divergence and recursion, with maximum data locality. It is not always clear how to “think” for the GPU, for instance, instead of the CPU, and most programming paradigms do not favor this style of “GPU thinking.”

Specifically, how do traditional compiler passes look if implemented with neither recursion nor branching control flow? Compiler transformations such as function lifting heavily modify the structure of the AST, making it non-trivial to convert traditional approaches into the “native vocabulary” of modern computing architectures, which reward parallel computation with vectorizable instructions over dense, compact structures.

This pearl describes two compiler passes, function lifting and variable anchoring, which can be understood through the following example, used throughout. This pseudo-scheme code describes a simple computation which contains a number of nested functions that capture variables at different lexical scopes.

(let ([m 3] [n 5])  
 (let ([f (lambda ()  
 (let ([y m] [z n])  
 ((lambda ()  
 (let ([x z])  
 ((lambda ()   
 (+ x y))))))))])  
 '()))

The function lifting and variable anchoring passes convert the above code to the following.

(letrec\*   
 ([env '((m 3) (n 5))]  
 [f (lambda (env\*)  
 (let ([env   
 `((y ,(get env\* 1 0))  
 (z ,(get env\* 1 1)))])  
 (fn1 (append env env\*))))]  
 [fn1 (lambda (env\*)  
 (let ([env `((x   
 ,(get env\* 1 1)))])  
 (fn2 (get env\* 1 0)  
 (append env env\*))))]  
 [fn2 (lambda (⍵ env\*)  
 (let ([env `()])  
 (+ (get env\* 1 0)  
 (get env\* 2 0))))]])  
 '())

Notice that all functions now appear at the top-level, that is, at the top letrec binding construct. Additionally, all variable references and bindings now take the form of references to or from an environment passed around to each function. For simplicity, the following treatment ignores flattening nested expressions and creating the closures for functions that may be returned by other functions, which would form other compiler passes.

|  |  |  |
| --- | --- | --- |
|  | Dyadic | Monadic |
| + | Addition | N/A |
| ⌈ | Max | N/A |
| , | Catenate | Vector of elements in row-major order (ravel) |
| ⍳ | Find first occurrence | Index Generate |
| = | Equality (of scalars) | N/A |
| ↑ | Take left elements from right | Mix (Nested vector converts to matrix) |
| ⊂ | N/A | Enclose argument as a rank 0 nested array |
| ∧ | Boolean AND | N/A |
| ∨ | Boolean OR | N/A |
| ⊢ | Right argument | Identity |
| ⍉ | N/A | Transpose |
| ⌽ | N/A | Reverse |
| ∩ | Intersection | N/A |
| ⍴ | N/A | Shape (vector of dimensions of the array) |
| ⊤ | Encode vector index to array index based on shape | N/A |
| ⊥ | Inverse of Encode, decode array index to vector index | N/A |

**Table 1.** Summary of Functions

The solution features concise and direct implementations of the core concepts behind function lifting and variable anchoring formulated with established array operations over n-dimensional rectangular arrays, giving a unique perspective on the traditional algorithmic descriptions of these core compiler passes. It clarifies the design of array-oriented compiler construction and lays out a roadmap for designing passes through consideration of the structures and encodings used to reason about the relationship between nodes in the AST. By utilizing standard array operations, the code highlights a specific set of array operations which prove generally useful for compiler transformations, providing a focal point for possible optimization and efficient implementation. All code may be executed using Dyalog APL 14.0 or later and is part of an effort to implement a high-performance, self-hosting array language compiler, Co-dfns, on the GPU and other parallel architectures.

* 1. Contributions
* A simple, direct, straightforward approach to function lifting and variable anchoring using only standard array operations
* The approach is defined in a readily available executable notation suitable for exploration and pedagogy
* The exposition suggests how further exploration of these algorithms might progress and how to explore the space of array programming to arrive at solutions to compiler transformations which might otherwise appear too difficult to implement
* The implementation highlights a set of general purpose operations that serve as focal points for future work to efficiently implement the algorithms described here

# Notational Conventions

All the algorithms in this pearl use a subset of APL's array notation suitable for execution on Dyalog APL 14.0 or later. All expressions evaluate from right to left with equal precedence order for all functions. Functions appear infix and take a right array argument with an optional left array argument. Functions that return other functions, called operators, take a left operand and possibly a right operand, either of which may be functions. Operators associate to the left instead of the right, as C expressions do. Thus, expressions associate as follows:

A f B g C ←→ A f (B g C)

Whereas all operators associate as follows:

f m n o ←→ ((f m) n) o

Finally, all operations occur over n-dimensional rectangular arrays. The representation of these arrays as dense or sparse does not matter here. Functions and operators taking two arguments or operands are called dyadic, while those taking only one are called monadic. ***Tables 1*** and ***2*** give the summary of functions and operators used.

A major cell is a sub-array of an array whose shape is that of the shape of the array less the first dimension. Thus, the major cells of a matrix are its rows. The rank 1 cells of a 3-dimensional array are the rows of each sub-matrix described by the array. The rank 2 cell of a matrix is just the matrix itself. In this treatment, scalar values are just rank 0 arrays. The concept of cells helps to explain how operators apply functions along arrays.

|  |  |
| --- | --- |
|  | Description |
| f/A | Reduce along last axis of A with function f |
| f⌿A | Reduce along first axis of A with function f |
| f⍀A | Scan along first axis of A with function f |
| A f.g B | Compute inner product of f and g over A and B |
| A ∘.f B | Compute the outer product of f over A and B |
| K f⌸ A | Apply f for each unique major cell in K with all associated cells in A |
| A f⍤i j⊢B | Compute f for each cell of rank i in A with cell of rank j in B |
| A f∘g B | Compose f with g, thus A f∘g B ←→ A f (g B) |

**Table 2.** Summary of Operators

The rank operator (⍤) generalizes a function to operate over cells. For instance, +/⍤1, computes the sum reduction of each row in a matrix, or each row of the sub-matrices in a 3-dimensional array, and so forth. The function ∘.f computes the outer product of f, which can be written using rank as follows:

∘.f ←→ f⍤0⍤0 99

Finally, some expressions use the convention known as function trains. The following equivalencies facilitate concise expression of functions. In the following, f, g, and h are functions, and A, B, and C are array values:

A (f g h) B ←→ (A f B) g (A h B)  
B (A g h) C ←→ A g (B h C)  
A (f g) C ←→ f (A g C)

# Language Definition

The compiled language consists of a simplified intermediate AST representing a program as might reasonably appear just before function lifting and flattening passes might occur. In particular, no function body contains any literal values. An attribute set attaches to each node in the AST, in addition to children that might appear. The following describes each node along with its set of attributes and its children:

Module[] := (FuncExpr | Expression)\*  
FuncExpr[name] := Function | Variable  
Function[] := Expression\*

Expression[name;class] :=   
 [Expression] FuncExpr Expression  
 | Variable  
Variable[name] := ()

All ASTs start with a single Module node as their sole root node. FuncExpr nodes encapsulate function objects either by reference or by definition. Functions contain expressions. Note that Function nodes model the same sort of functions described in the previous section on notation. They receive a single right argument and a single, optional left argument. Expressions either refer to a variable binding or one of two sorts of function application. The class attribute of the Expression indicates the expression type, either a variable reference, a monadic application of a function, or a dyadic application. The FuncExpr and Expression nodes may contain a name attribute, indicating a binding for that particular object. This example from the introduction looks like this when parsed:

M  
┌─────┬──┴──┬───────┐  
E[m←] E[n←] E[nil←] FE[f←]  
│ │ │ │  
N[3] N[5] N[0] F  
 ┌─────┼─────┐  
 E[y←] E[z←] E  
 │ │ ├──┐  
 V[m] V[n] FE E  
 │ │  
 F V[nil]  
 ┌──┴──┐  
 E[x←] E  
 │ ├──┐  
 V[z] FE E  
 │ │  
 F V[nil]  
 │  
 E  
 ┌────┼────┐  
 E FE E  
 │ │ │  
 V[x] P[+] V[y]

In addition to the explicit restrictions on the AST, the compiler passes described here presume that a dead-code elimination pass has already removed useless or dead code.

* 1. Encoding the AST

Because all the compiler passes operate over arrays, the AST must exist in some array encoding of the tree structure, in this case, a matrix. Each row corresponds to a single node in the AST, ordered by pre-order depth-first traversal. The first column contains the depth vector of the nodes in the AST, starting with 0 for the root node. The second column contains the name of the node. The other columns contain attributes for the nodes. The attributes on input are name and class, columns 3 and 4, respectively. This representation encodes all relevant information, but also has the added benefit of being a standard tree encoding used within the APL programming community. *Throughout, some expressions begin with* ⍉*, the transpose function, to swap the rows and columns in order to display more neatly. In such cases, note that the rows run vertically and the columns run horizontally.* ***Snippet 1*** shows how the example given in the introduction looks encoded.

**Snippet 1.** Initial AST transposed to display more nicely

⍉AST

┌─┬─┬─┬─┬─┬───┬─┬──┬─┬─┬─┬─┬─┬─┬──┬─┬─┬─┬─┬──┬─┬─┬──┬──┬──┬──┬──┬──┬─┬───┬─┬───┐

│0│1│2│1│2│1 │2│1 │2│3│4│3│4│3│4 │5│6│7│6│7 │8│9│10│11│10│11│10│11│7│8 │4│5 │

├─┼─┼─┼─┼─┼───┼─┼──┼─┼─┼─┼─┼─┼─┼──┼─┼─┼─┼─┼──┼─┼─┼──┼──┼──┼──┼──┼──┼─┼───┼─┼───┤

│M│E│N│E│N│E │N│FE│F│E│V│E│V│E│FE│F│E│V│E│FE│F│E│E │V │FE│P │E │V │E│V │E│V │

├─┼─┼─┼─┼─┼───┼─┼──┼─┼─┼─┼─┼─┼─┼──┼─┼─┼─┼─┼──┼─┼─┼──┼──┼──┼──┼──┼──┼─┼───┼─┼───┤

│ │m│3│n│5│nil│0│f │ │y│m│z│n│ │ │ │x│z│ │ │ │ │ │x │ │+ │ │y │ │nil│ │nil│

├─┼─┼─┼─┼─┼───┼─┼──┼─┼─┼─┼─┼─┼─┼──┼─┼─┼─┼─┼──┼─┼─┼──┼──┼──┼──┼──┼──┼─┼───┼─┼───┤

│ │a│ │a│ │a │ │ │ │a│ │a│ │m│ │ │a│ │m│ │ │d│a │ │ │ │a │ │a│ │a│ │

└─┴─┴─┴─┴─┴───┴─┴──┴─┴─┴─┴─┴─┴─┴──┴─┴─┴─┴─┴──┴─┴─┴──┴──┴──┴──┴──┴──┴─┴───┴─┴───┘

The full names and attributes have obviously been adjusted to make the picture prettier.

# Compiler Passes

In order to perform flattening and lifting, the following series of compiler passes suffice to move from an AST as described above to the following flattened AST:

Module[] := (FuncExpr | Expression)\*  
FuncExpr[name] := Function | Variable  
Function[] := Expression\*  
Variable[name] := ()  
Expression[name;class;left;right;fn;  
 left\_env;env;slot;left\_slot;  
 right\_env;right\_slot] := ()

In particular, all function bindings now occur at the top-level, and all function bodies contain expressions with no children. Instead, each expression contains attributes for the left and right arguments as well as the function name. With the name, for each reference there are two additional attributes, giving the environment index and slot position for that variable's location. The environment indicates how far up in the lexical stack to reach, and the slot gives the appropriate position in that environment that contains the variable referenced.

For simplicity this section omits expression flattening, since the techniques match those of function lifting.

* 1. Node Coordinates

Traditional implementations of lifting and flattening encode information about the structure of the AST through recursion and branching on node types. Any method of lifting and flattening must encode this information in some usable way. Instead of using recursion and control flow to encode this structural information, consider the depth vector D of the AST. This vector encodes all of the structural information required to understand the parent-child relationships of the AST. Unfortunately, it does so in a way that does not allow for local reasoning about any two nodes in the tree without surrounding context.

Before function lifting, a compiler pass re-encodes this information such that the structural relationship of any two nodes exists locally for each node, requiring no external context. In particular, given the reference or coordinate for any two nodes, simple array operations suffice to determine whether one node is an ancestor of the other, the depth of each node in relation to the other, and whether one node appears before or after the other in the traversal. This information will prove critically important in other passes, so the pass annotates each node with an additional ref attribute containing the node reference, also called a coordinate.

A node is an ancestor of another node if and only if its coordinate is a prefix of the second node's coordinate, ignoring zeros. A coordinate corresponds to a path from the root of the tree to the node. It is a vector of length equal to the depth of the AST, consisting of natural numbers, whose non-zero elements precede its zero elements, the count of which equal the depth of the node. A coordinate also represents a unique identifier for any given node as an index into a multi-dimensional array whose rank is the depth of the AST.

To compute the matrix of all coordinates, consider first the set of natural numbers from zero to the depth of the AST, inclusive, given by the following expression:

⍳1+⌈/0,D  
0 1 2 3 4 5 6 7 8 9 10 11

Here D is the depth vector of the AST. Now consider the boolean matrix given by the outer product equating each element of the range and each element of the original depth vector, given by the first expression in ***Snippet 2***.

**Snippet 2.** Visualizing AST using Outer Product; using Scan to obtain unique identifiers; adjusting to have node coordinates

The result depicts a neat little pictorial representation of the information encoded in the depth vector, using more space. In particular, the tree like structure becomes obvious. The structure defined by this tree becomes more clear by the outer product.

This picture suggests another way of encoding the same information by scanning along the first dimension, resulting in the prefix sum for each column of the matrix. The second expression demonstrates this result using the same depth vector.

⍉D∘.=⍳1+⌈/0,D

**1** 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 **1** 0 **1** 0 **1** 0 **1** 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 **1** 0 **1** 0 **1** 0 **1** 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 **1** 0 **1** 0 **1** 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 **1** 0 **1** 0 **1** 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 **1** 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 **1** 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 **1**

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 **1** 0 **1** 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 **1** 0 **1** 0 0 0 0 0 0 0 0 **1** 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 **1** 0 0 0 0 0 0 0 0 **1** 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 **1** 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 **1** 0 **1** 0 **1** 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 **1** 0 **1** 0 **1** 0 0 0 0

⍉+⍀D∘.=⍳1+⌈/0,D

1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

0 1 1 2 2 3 3 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4

0 0 1 1 2 2 3 3 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4

0 0 0 0 0 0 0 0 0 1 1 2 2 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3

0 0 0 0 0 0 0 0 0 0 1 1 2 2 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 4 4

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 2 2 2 2 2 2 2 2 2 3 3 3 3

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 2 2 2

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 2 2 3 3 3 3 3 3

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 2 2 3 3 3 3 3

⍉R←(1+D)↑⍤¯1+⍀D∘.=⍳1+⌈/0,D

1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

0 1 1 2 2 3 3 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4

0 0 1 0 2 0 3 0 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4

0 0 0 0 0 0 0 0 0 1 1 2 2 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3

0 0 0 0 0 0 0 0 0 0 1 0 2 0 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 4 4

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 2

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 2 2 2 2 2 2 2 2 2 2 2 2 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 2 2 2 2 2 2 2 2 2 3 3 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 0 2 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 2 2 3 3 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 2 0 3 0 0 0 0

Now, each row has a unique value encoding an index, and nearly all the desired invariants exist in the above matrix. However, spurious non-zero values exist which do not contribute to the uniqueness of the coordinates and likewise provide no further useful information. These spurious values are any non-zero values that appear in columns past the depth of any given node. By taking only the number of non-zero values up to the depth of each node, the appropriate coordinate matrix emerges, as seen in the third expression of ***Snippet 2***.

Given this coordinate matrix, named R, all the other compiler passes may now compute with the parent-child relationships without requiring recursion or branching.

* 1. Function Lifting

Function lifting involves three particular insights to complete. Firstly, note that the coordinate matrix describes a finite multi-dimensional space, with each coordinate forming a point somewhere in this space. The size of this space is given by the range of each column in the matrix, that is, the maximum value plus one for each column. A node coordinate represents a unique identifier. The following expression computes the size of each dimension of this space, called RM:

⊢RM←1+⌈⌿R  
2 5 5 4 5 3 3 4 3 2 4 4

Here R represents the coordinate matrix. Given RM and any row of R, the vector index given by the row may be linearized into a unique integer index for that node, given by RM⊥⍺ where ⍺ is a row of R.

RM⊥1 4 4 2 0 0 0 0 0 0 0 0  
3421440

Each node stores this information locally, so each lifted Function node already provides a unique variable with which to replace it in the tree. This allows all variable generation to occur without requiring an accumulator or some other stateful system, which might require non-local computation.

Next, some general strategy must exist for doing the actual lifting. In this case, the answer comes from a surprising place. Assume that a function ngh takes a coordinate as its left argument and a matrix group of nodes as its right. In this case the coordinate is the coordinate of the function directly enclosing the nodes given on the right. The group of nodes in the right argument is the body of the function, including all Function nodes appearing in the body of the function, but *without* their bodies. In this case, the ngh function has all the information necessary to create the lifted function definition at the top level. It performs the following operations:

1. It replaces each Function node, which no longer has children, with a Variable node using the unique identifier derived from the coordinate given in the ref attribute of the node;
2. Manipulates the depth vector of the group to shift all nodes to the top-level depth, preserving order and internal relationships, and finally;
3. Adds a new function definition node enclosing the function group, using the given coordinate it received as its left argument.

Note the importance of reusing the coordinate after lifting. The coordinate information must preserve the original structural information of the code, even after compiler passes mostly remove this structure from the AST. The appendix contains a complete definition of the ngh function for the curious.

Now assume that there exists a value C which contains the coordinate for each node of the tree that corresponds to the Function to which that node belongs. This C matrix contains the grouping information for each function body, in the form of a set of keys by which to group each node. At this point, all of the information is in place to do function lifting, if only a strategy for lifting existed. The array programming community possesses just such a strategy. The Key (written ⌸) operator existed as an array programming primitive since at least the J programming language and the Connection Machine [Hui 2014]. Given an array representation of keys on the left and a set of corresponding elements on the right, it applies its left operand once for each unique key in the left argument, passing the unique key as the left argument to the operand, and the set of elements associated with that key as the right argument. Here is a simple example:

2 1 3 2 3 ,∘⊂⌸ ⍳5  
┌─┬───┐  
│2│0 3│  
├─┼───┤  
│1│1 │  
├─┼───┤  
│3│2 4│  
└─┴───┘

Importantly, the key operator preserves the order both of the appearance of the keys as well as the appearance of the elements. By applying this key operator to the set of scope keys given by C and the operand ngh, the result is a re-ordered tree with each scope given its own top-level function definition. This also has the side-effect of lifting all top-level expressions into a single group, which turns out to be useful in future passes. Thus, given C and ngh, the following expression lifts all functions to the top level.

C ngh⌸ ⍵

Here ⍵ is the body of the Module.

How does one compute C? Firstly, compute the value SC, which is all the scope coordinates with their last non-zero element zeroed. Secondly, compute RF, the coordinate matrix of all scope nodes, namely Module and Function. Then, the code in ***Snippet 3*** computes the boolean matrix indicating which elements of RF are prefixes or equal to each row in SC by using an inner product.

The function (=∨0=⊢) is a predicate indicating whether the right argument is a prefix or equal to the left argument.

**Snippet 3.** Computing SC, RF, and the Scope Mask

1 4 4 0 0(=∨0=⊢)1 4 0 0 0  
1 1 1 1 1  
 1 4 3 0 0(=∨0=⊢)1 4 0 0 0  
1 1 1 1 1  
 1 2 4 0 0(=∨0=⊢)1 4 0 0 0  
1 0 1 1 1

⍉SC

1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

0 1 0 2 0 3 0 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4

0 0 0 0 0 0 0 0 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4

0 0 0 0 0 0 0 0 0 1 0 2 0 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3

0 0 0 0 0 0 0 0 0 0 0 0 0 0 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 0 4

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 2 2 2 2 2 2 2 2 2 2 2 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 2 2 2 2 2 2 2 0 3 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 2 0 3 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

RF

1 0 0 0 0 0 0 0 0 0 0 0

1 4 4 0 0 0 0 0 0 0 0 0

1 4 4 3 3 1 0 0 0 0 0 0

1 4 4 3 3 1 2 2 1 0 0 0

⍉SC∧.(=∨0=⊢)⍉RF

1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 0 0 0 0

This is combined with boolean ∧ in an inner product given by ∧.(=∨0=⊢) to compute the boolean matrix.

By ordering SC and RF lexicographically, the last true value in each row of the resulting matrix indicates the closest enclosing scope for each node. From this information the C matrix can be constructed naturally.

For simplicity, this treatment omits an extended exposition of ngh, as it is a natural function that requires only simple manipulation in an obvious and direct way to achieve its results. The curious may observe the full definition of ngh in the appendix.

The above technique using the Key operator for flattening also works when applied to nested expressions.

* 1. Variable Anchoring

Variable anchoring converts variable references or assignments into a lexical address, indicating the environment in which to find that variable, as well as the “slot” or position in which that variable occurs within the environment. This removes any environment lookups for all subsequent passes in the compiler. At the point where variable anchoring occurs, the AST already exhibits a mostly flat structure, with all functions at the top level and all expressions flattened out to a linear form. Again, a handler for each scope or function body may operate independently of each other scope through the use of the key operator. Note that it is not necessary to use the node coordinates for grouping at this point, as the depth vector provides sufficient information to group using a simple sum scan.

The handler does require two structures for its operation. The SK matrix gives the node coordinates for each scope node, indicating the total set of all possible environments in the module. The SB matrix gives the bindings for each of these scopes. Each row in SB contains a set of the unique names bound in the scope identified by the corresponding row in SK.

SB  
┌─┬─┬───┐  
│m│n│nil│  
├─┼─┼───┤  
│y│z│ │  
├─┼─┼───┤  
│x│ │ │  
├─┼─┼───┤  
│ │ │ │  
└─┴─┴───┘

Taking r to be the coordinate of the currently processed scope, the prefixes of r which appear in SK determine the relevant rows out of SB that may contribute to the bindings in this scope, given by the following expression:

(⍉av fe lf rn AST)[(⍳7),9+⍳5;]

┌─┬─┬─┬─┬─┬───┬─┬──┬─────────┬─────────┬─┬─┬─┬─────────┬─────────┬─┬─┬─────────┬─────────┬─┬─┐

│0│1│2│1│2│1 │2│1 │2 │1 │2│3│3│3 │1 │2│3│3 │1 │2│3│

├─┼─┼─┼─┼─┼───┼─┼──┼─────────┼─────────┼─┼─┼─┼─────────┼─────────┼─┼─┼─────────┼─────────┼─┼─┤

│M│E│N│E│N│E │N│FE│V │FE │F│E│E│E │FE │F│E│E │FE │F│E│

├─┼─┼─┼─┼─┼───┼─┼──┼─────────┼─────────┼─┼─┼─┼─────────┼─────────┼─┼─┼─────────┼─────────┼─┼─┤

│ │m│3│n│5│nil│0│f │fn3386880│fn3386880│ │y│z│ │fn3450240│ │x│ │fn3451232│ │ │

├─┼─┼─┼─┼─┼───┼─┼──┼─────────┼─────────┼─┼─┼─┼─────────┼─────────┼─┼─┼─────────┼─────────┼─┼─┤

│ │ │ │ │ │ │ │ │ │ │ │ │ │ │ │ │ │ │ │ │x│

├─┼─┼─┼─┼─┼───┼─┼──┼─────────┼─────────┼─┼─┼─┼─────────┼─────────┼─┼─┼─────────┼─────────┼─┼─┤

│ │ │ │ │ │ │ │ │ │ │ │ │ │fn3450240│ │ │ │fn3451232│ │ │+│

├─┼─┼─┼─┼─┼───┼─┼──┼─────────┼─────────┼─┼─┼─┼─────────┼─────────┼─┼─┼─────────┼─────────┼─┼─┤

│ │ │ │ │ │ │ │ │ │ │ │m│n│nil │ │ │z│nil │ │ │y│

├─┼─┼─┼─┼─┼───┼─┼──┼─────────┼─────────┼─┼─┼─┼─────────┼─────────┼─┼─┼─────────┼─────────┼─┼─┤

│ │l│ │l│ │l │ │ │ │ │ │a│a│m │ │ │a│m │ │ │d│

├─┼─┼─┼─┼─┼───┼─┼──┼─────────┼─────────┼─┼─┼─┼─────────┼─────────┼─┼─┼─────────┼─────────┼─┼─┤

│0│0│0│1│0│2 │0│0 │0 │0 │0│0│1│2 │0 │0│0│1 │0 │0│0│

├─┼─┼─┼─┼─┼───┼─┼──┼─────────┼─────────┼─┼─┼─┼─────────┼─────────┼─┼─┼─────────┼─────────┼─┼─┤

│0│0│0│0│0│0 │0│0 │0 │0 │0│0│0│0 │0 │0│0│0 │0 │0│1│

├─┼─┼─┼─┼─┼───┼─┼──┼─────────┼─────────┼─┼─┼─┼─────────┼─────────┼─┼─┼─────────┼─────────┼─┼─┤

│0│0│0│0│0│0 │0│0 │0 │0 │0│2│2│2 │0 │0│1│1 │0 │0│0│

├─┼─┼─┼─┼─┼───┼─┼──┼─────────┼─────────┼─┼─┼─┼─────────┼─────────┼─┼─┼─────────┼─────────┼─┼─┤

│0│0│0│0│0│0 │0│0 │0 │0 │0│1│1│1 │0 │0│1│2 │0 │0│2│

├─┼─┼─┼─┼─┼───┼─┼──┼─────────┼─────────┼─┼─┼─┼─────────┼─────────┼─┼─┼─────────┼─────────┼─┼─┤

│0│0│0│0│0│0 │0│0 │0 │0 │0│0│1│2 │0 │0│1│2 │0 │0│0│

└─┴─┴─┴─┴─┴───┴─┴──┴─────────┴─────────┴─┴─┴─┴─────────┴─────────┴─┴─┴─────────┴─────────┴─┴─┘

**Snippet 4.** Final AST

↑SK  
1 0 0 0 0 0 0 0 0 0 0 0  
1 4 4 0 0 0 0 0 0 0 0 0  
1 4 4 3 3 1 0 0 0 0 0 0  
1 4 4 3 3 1 2 2 1 0 0 0  
 r  
1 4 4 0 0 0 0 0 0 0 0 0  
 ⌽1+⍳r⍳0  
3 2 1  
 (⌽1+⍳r⍳0)((≢⊢)↑↑)⍤0 1⊢r  
1 4 4 0 0 0 0 0 0 0 0 0  
1 4 0 0 0 0 0 0 0 0 0 0  
1 0 0 0 0 0 0 0 0 0 0 0  
 ↑(↓(⌽1+⍳r⍳0)((≢⊢)↑↑)⍤0 1⊢r)∩SK  
1 4 4 0 0 0 0 0 0 0 0 0  
1 0 0 0 0 0 0 0 0 0 0 0  
 SK⍳(↓(⌽1+⍳r⍳0)((≢⊢)↑↑)⍤0 1⊢r)∩SK  
1 0

These rows form the b matrix for the current scope, which is used to find the slots and environments for each name appearing in the given scope, given by nm. This can be done quite easily because an index into b pointing to any given variable is the same as the environment and slot for that variable. Thus, it suffices to find the index of each variable relative to b, which the following expression accomplishes through the use of the find (⍳) function and the encode (⊤) function.

nm  
┌─┬┬─┬─┬┬─┬┬┬───┐  
│y││m│z││n│││nil│  
└─┴┴─┴─┴┴─┴┴┴───┘  
 b  
┌─┬─┬───┐  
│y│z│ │  
├─┼─┼───┤  
│m│n│nil│  
└─┴─┴───┘  
 (⍴b)⊤(,b)⍳nm  
0 0 0 0 0 0 0 0 1  
0 2 0 0 2 0 2 2 2

Note here that some additional care must be used depending on how the scoping in a given language works. For example, a variable could be bound in one scope, but may be bound later in the function body, such that references to the same variable before the binding actually refer to a binding in a containing environment. The techniques described above still apply, but require some additional book-keeping to ensure that incorrect bindings do not occur. Bindings at the local scope in b must be adjusted or masked to avoid matching against local bindings which are bound later than the resolved reference.

* 1. Putting it together

Given the initial ast, and a pass which does flattening of the expressions, fe, the three passes described above, rn, lf, and av, respectively, combine to transform the AST from it's original form to its final flattened form, given in ***Snippet 4*** without the ref attribute for space.

# Related Work

The J programming language [Hui 2014] was the first practical, general-purpose programming language to introduce the key operator as a primitive operator with the presumption of its general usefulness. The rank operator (⍤) used throughout this treatment also derives from the J traditional, receiving particular interest throughout the APL community [Bernecky 1987; Hui 1995].

The EigenCFA effort [Prabhu 2011] demonstrated significant performance improvements of a 0-CFA flow analysis by utilizing similar techniques to those demonstrated here. In particular, encoding the AST and using accessor functions have a very similar feel to the node coordinates and AST encoding given here, though they have a different formulation and spend considerable effort understanding the trade-offs of performance associated with the different encodings, whereas the encodings here were chosen for their clarity and directness, rather than their performance.

Mendez-Lojo, et al. implemented a GPU version of Inclusion-based Points-to Analysis [Mendez-Lojo 2012] that also focuses on adapting data structures and algorithms to efficiently execute on the GPU. In particular, they use similar techniques of prefix sums and sorts to achieve some of their adaptation to the GPU, Additionally, they have clever and efficient methods of representing graphs on the GPU which enable dynamic rewriting of the graph.

The APEX compiler [Bernecky 1997] developed vectorized approaches to handling certain analyses to compile traditional APL, including a SIMD tokenizer [Bernecky 2003]. It uses a SSA representation, and converts the dynamic scope of traditional APL functions into a static form early on. It also uses a matrix format to represent the AST. Traditional APL did not have nested function definitions, however, and thus the APEX compiler does not have any specific approaches to dealing with function lifting.

Bernecky further identified methods of reducing or optimizing the computational complexity or cost of certain array operations, allowing improved performance of easy to understand array expressions [Bernecky 1999].

Timothy Budd implemented a compiler [Budd 1984, 1988] for APL which targeted vector processors as well as one for C code. They used a method of lazy evaluation to avoid intermediate data copying. Budd provided thoughts and some ideas on how the compiler might be implemented in parallel as well.

Walter Schwarz implemented an APL to C compiler for the ACORN system targeting the CM-2 machine, demonstrating performance potential for APL as a massively parallel language [Schwarz 1991].

W. Ching and D. Ju have spent significant work on the ELI language and other APL-class language implementations, especially on parallelized code and optimization.[Ching 1990, 1993, 1994, 2000; Ju 1991].

J. D. Bunda and J. A. Gerth presented a method for doing table driven parsing APL which suggested a parallel optimization for parsing, but did not elucidate the algorithm [Bunda 1984].

# Future Work

A small compiler for a lexically scoped dialect of APL (Co-dfns) currently utilizes these approaches to implement the lexical features of the language. It targets Cuda and C for further compilation. All the given code above elucidates the concepts as directly as possible, without significant consideration to the target upon which they will execute.

The efficient execution of the above expressions on vector-oriented architectures such as GPUs, or large distributed machines, will require consideration of the array representations such as whether to use dense or sparse matrices (evidence suggests sparse matrices are a clear win [Prabhu 2011]). Additionally, efficient implementations for operators such as inner product and key must exist for the target platforms.

Further exploration of the compiler design will determine whether these techniques scale to other problems, such complicated optimizations. Prior research indicates that techniques such as these do in fact enable efficient implementation on GPUs [Prabhu 2011], but requires more study of different algorithms and optimizations, as well as a more complex language than that presented here.

The above techniques suggest that a self-hosted APL compiler that efficiently executes on the GPU does not exceed the realm of reasonable expectation. Moreover, they suggest that a fully vectorized compiler may not require each pass to exhibit unique cleverness, but that some core techniques may provide sufficient power to implement many of the passes, enabling a focusing of optimization effort to the core primitives used to implement the passes, such as rank and key. Significant work still remains to accomplish such a goal, however.

# Conclusion

Given a few key insights into encoding a tree into an AST and representing relationships between nodes in a suitable manner, one can implement lexical scoping semantics for a given language using nothing but well known array operations, without complex control flow, recursion, or branching. The rank, scan, key, inner and outer product operators form the backbone of such computation, suggesting that efficient implementation of these primitives may enable a completely self-hosted array language compiler to exist on the GPU in the future. Using a suitable notation and approach may also reduce the overheads required to create compiler passes which exhibit favorable characteristics for modern architectures, making the study and comprehension of such algorithms more accessible than previously.

# Acknowledgments

This work is graciously funded by Dyalog, Ltd. to support the construction of a high-performance APL compiler. Bob Bernecky, Roger Hui, Eric Holk, and Ed Amsden all provided useful feedback on early drafts.

References

Robert Bernecky. 1987. An introduction to function rank. SIGAPL APL Quote Quad 18, 2 (December 1987), 39-43. DOI=10.1145/377719.55632 http://doi.acm.org/10.1145/377719.55632

Robert Bernecky. 1997. "APEX: The APL parallel executor." (1997).

Robert Bernecky. 1999. "Reducing computational complexity with array predicates." ACM SIGAPL APL Quote Quad 29, no. 3 (1999): 39-43.

Robert Bernecky. 2003. "An SPMD/SIMD parallel tokenizer for APL." In Proceedings of the 2003 conference on APL: stretching the mind, pp. 21-32. ACM, 2003.

Timothy A. Budd. 1988. An APL compiler. NY: Springer-Verlag, 1988.

Timothy A. Budd. 1984. An APL Compiler for a Vector Processor. ACM Trans. Program. Lang. Syst. 6, 3 (July 1984), 297-313. DOI=10.1145/579.357248 http://doi.acm.org/10.1145/579.357248

J. D. Bunda and J. A. Gerth. 1984. APL two by two-syntax analysis by pairwise reduction. SIGAPL APL Quote Quad 14, 4 (June 1984), 85-94. DOI=10.1145/384283.801081 http://doi.acm.org/10.1145/384283.801081

Wai-Mee Ching. 2000. The design and implementation of an APL dialect, ELI. In Proceedings of the international Conference on Apl-Berlin-2000 Conference (Berlin, Germany, July 24 - 27, 2000). APL '00. ACM, New York, NY, 69-76. DOI= http://doi.acm.org/10.1145/570475.570485

Wai-Mee Ching and A. Katz. 1994. An experimental APL compiler for a distributed memory parallel machine. In Proceedings of the 1994 ACM/IEEE Conference on Supercomputing (Washington, D.C., November 14 - 18, 1994). IEEE Computer Society Press, Los Alamitos, CA, 59-68.

Wai-Mee Ching, P. Carini, and D. Ju. 1993. A primitive-based strategy for producing efficient code for very high level programs. Comput. Lang. 19, 1 (Jan. 1993), 41-50. DOI= http://dx.doi.org/10.1016/0096-0551(93)90038-3

Wai-Mee Ching. 1990. Automatic parallelization of APL-style programs. In Conference Proceedings on APL 90: For the Future (Copenhagen, Denmark, August 13 - 17, 1990). P. Gjerløv, Ed. APL '90. ACM, New York, NY, 76-80. DOI= http://doi.acm.org/10.1145/97808.97826

F. Hendriks and Wai-Mee Ching. 1990. Sparse matrix technology tools in APL. In Conference Proceedings on APL 90: For the Future (Copenhagen, Denmark, August 13 - 17, 1990). P. Gjerløv, Ed. APL '90. ACM, New York, NY, 186-191. DOI= http://doi.acm.org/10.1145/97808.97844

D. Ju, Wai-Mee Ching, and C. Wu. 1991. On performance and space usage improvements for parallelized compiled APL code. In Proceedings of the international Conference on APL '91 (Palo Alto, California, USA, August 04 - 08, 1991). APL '91. ACM, New York, NY, 234-243. DOI= http://doi.acm.org/10.1145/114054.114080

D. Ju. and Wai-Mee Ching. 1991. Exploitation of APL data parallelism on a shared-memory MIMD machine. In Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (Williamsburg, Virginia, USA, April 21 - 24, 1991). PPOPP '91. ACM, New York, NY, 61-72. DOI= http://doi.acm.org/10.1145/109625.109633

Roger Hui. 1995. Rank and uniformity. In Proceedings of the international conference on Applied programming languages (APL '95), Marc Griffiths and Diane Whitehouse (Eds.). ACM, New York, NY, USA, 83-90. DOI=10.1145/206913.206968 http://doi.acm.org/10.1145/206913.206968

Roger Hui. 2014. Essays/Key. http://www.jsoftware.com/jwiki/Essays/Key

Mario Mendez-Lojo, Martin Burtscher, and Keshav Pingali. 2012. A GPU implementation of inclusion-based points-to analysis. In Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming (PPoPP '12). ACM, New York, NY, USA, 107-116. DOI=10.1145/2145816.2145831 http://doi.acm.org/10.1145/2145816.2145831

Walter Schwarz. 1991. "Acorn Run-Time System for the CM-2." In Arrays, Functional Languages, and Parallel Systems, pp. 35-57. Springer US, 1991.

Tarun Prabhu, Shreyas Ramalingam, Matthew Might, and Mary Hall. 2011. EigenCFA: accelerating flow analysis with GPUs. In Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '11). ACM, New York, NY, USA, 511-522. DOI=10.1145/1926385.1926445 http://doi.acm.org/10.1145/1926385.1926445

*Complete pass definitions*

┌───────────────────────────────────────────┬──────────────────────────────────────┐

│ E←{(1⌷⍉⍵)∊⊂,'E'} │ F←{(1⌷⍉⍵)∊⊂,'F'} │

├───────────────────────────────────────────┼──────────────────────────────────────┤

│ lf←{ │ av←{ │

│ rm←1+⌈⌿r←↑4⌷⍉⍵ │ sb←(k←+\1⌽F ⍵){2⌷⍉(E ⍵)⌿⍵}⌸⍵ │

│ sc←1↓(≢↑¯1↓(0≠⊢)(/∘⊢)⊢)⍤1⊢r │ sk←(1,1↓F ⍵)/7⌷⍉⍵ │

│ rf←(1,1↓F ⍵)⌿r │ ⊃⍪/k(sk sh sb)⌸⍵ │

│ c←(⌈/(⍳≢rf)×⍤1⊢sc∧.(=∨0=⊢)⍉rf)⌷⍤0 2⊢rf│ } │

│ (1↑⍵)⍪⊃⍪/c(rm ngh)⌸1↓⍵ │ │

│ } │ │

├───────────────────────────────────────────┼──────────────────────────────────────┤

│ ngh←{ │ sh←{ │

│ rf←⍺⍺{(≢⍺⍺)↑⊃¯1↑⍵}⍤1⊢(fn←F ⍵)⌿g←⍵ │ r←(0≠⍺)⊃7⌷⍉⍵ │

│ (2⌷⍉fn⌿g)←(⊂'fn'),∘⍕¨⍺⍺⊥⍉rf │ i←⍺⍺⍳⍺⍺∩⍨↓(⌽1+⍳r⍳0)((≢⊢)↑↑)⍤0 1⊢r│

│ (1⌷⍉fn⌿g)←⊂,'V' │ b←i⌷⍤0 2⊢⍵⍵ │

│ ⊂⍺(⍺⍺ fnh)⍣(1≠+/⍺)⊢g │ nm←((e←E ⍵)⌿⍵)[;2 3 5] │

│ } │ ⊂⍵,e⍀((≢nm),6)⍴⍉(⍴b)⊤(,b)⍳,nm │

│ │ } │

├───────────────────────────────────────────┼──────────────────────────────────────┤

│ fnh←{ │ rn←{⍵,↓(1+d)↑⍤¯1+⍀d∘.=⍳1+⌈/0,d←0⌷⍉⍵} │

│ h←⍉⍪1 'FE'('fn',⍕⍺⍺⊥⍺)''(0⍴⍨≢⍺⍺) │ │

│ h⍪←2(,'F')'' ''⍺ │ │

│ h⍪({⍵-(⊃⍵)-3}0⌷⍉⍵),1↓⍤1⊢⍵ │ │

│ } │ │

└───────────────────────────────────────────┴──────────────────────────────────────┘