The Key to a Data-parallel Compiler

A Functional Pearl

|  |
| --- |
| Aaron W. Hsu  Indiana University  awhsu@indiana.edu |

By choosing an appropriate representation for the AST, we can implement a compiler for a modern functional language using only data-parallel array operations under simple functional composition without the use of branching, recursion, or other means of explicit control flow. In this pearl we elucidate the core data structure and an operator, called Key, that enable a surprisingly direct implementation of non-trivial transformations. The result permits arbitrary computation over sub-trees chosen by parent-child and other inter-node relationships in a pure data-parallel fashion without complex control flow. We give direction on the use of this method through examples of function lifting and expression flattening.

Categories and Subject Descriptors D.3.4 [**Programming Languages**]: Processors – compilers.

General Terms Algorithms, Design, Languages.

Keywords compiler, data-parallel, data-flow, key operator, node coordinates, array programming, APL

1. Introduction

When compiling a high-level language into a low-level language, we often implement a variety of flattening transformations to normalize more complex code exhibiting nesting into flat programs that are semantically equivalent. Function lifting, for example, takes a program with nested functions into an equivalent program in which all functions are at the root of the program. The following function *f* always returns 14.

f←λ.(λ ⍵.(⍵+⍵))7

A function lifting pass would transform this into the following:

fn1←λ⍵.⍵+⍵  
f←λ.fn1 7

Expression flattening works in a similar fasion. Assuming all expressions evaluate right to left, this expression:

v←(a+b)÷c×d

Becomes this:

v←c×d  
t←a+b  
v←t÷v

Characteristic of many compiler passes, most of the work is manipulating the tree, rather than, say, number crunching. Almost universally, compilers use a combination of recursion and branching to implement these transformations, such as using the visitor pattern, Nanopass framework [19], or direct structural recursion. The “work” happens in the control flow of the program text. The ubiquity of this approach warrants note. Let us savour such rare solidarity for a moment. Now, let us challenge it.

Let us arbitrarily restrict ourselves to a data-flow, data-parallel model. Specifically, consider a language whose core primitives are pointwise functions over *n*-dimensional arrays. We have a set of operators that take functions and apply them over different sub-parts of arrays. We can compose these functions together. That is all. Can we implement a compiler in such a language? The obvious answer is, yes, given arbitrary primitives. What if we restrict ourselves to standard and general-purpose primitives? The answer is still yes. Furthermore, the method is surprisingly direct and concise. The language we use is a subset of Dyalog APL 14.0 and the reader may follow the examples in the following sections using TryAPL.org.

The trick to such a compiler is solving the issue illustrated by the previous examples, expressing tree manipulation without embedding the core logic in the control flow. We accomplish this by choosing an appropriate representation of the AST and a property of each node we call its *node coordinate*. These coordinates allow us to locally answer the inter-node questions we need without complex control flow. When we combine this with a general-purpose operator called “Key,” we can perform arbitrary computation over sub-trees that we partition based on these inter-node relationships, such as whether one node is an ancestor of another. When combined with more mundane array programming, we can implement the complete core of a compiler, including lexical scope, function lifting, expression flattening/normalization, loop fusion, frame/register allocation, and so on. The result is a completely data-parallel compiler modulo parsing and some aspects of code generation.

The notation used throughout this paper is introduced piecemeal as necessary in each relevant section in which the notation is first used. All functions are applied infix and take either one or two arguments. All expressions evaluate from right to left. Tables 1 and 2 provide a reference for the notation and primitives as a reference aid.

* 1. Contributions
* We introduce node coordinates as a means to locally compute inter-node properties without using complex control flow.
* We demonstrate methods to perform arbitrary computation over sub-trees partitioned by these inter-node properties in a pure data-flow, data-parallel style.
* The presented methods are direct and concise; the data structures, simple.
* These methods require no specialized operations, but instead, use only established, general-purpose array programming.

1. A Definition of Node Coordinates

Every node in an AST has a node coordinate, named because a coordinate is a precise location in a space. We can imagine all the nodes arranged inside some multi-dimensional space, leading to a specific set of node coordinates. Many such arrangements exist, of varying usefulness. In our case, any arrangement should allow us to answer the following questions about any two arbitrary nodes in a tree:

1. Are the nodes the same?
2. Are they siblings?
3. Does one appear “earlier” in the tree?
4. Are they at the same depth?
5. Is one an ancestor of another?

In short, we care about the relative position of each node in a tree relative to any other. We define a node coordinate as follows.

A *node coordinate* is a vector whose length is the depth of the tree. Its elements are natural numbers. The count of non-zero elements in the vector is equal to the depth of that node in the tree. All zero elements appear after the non-zero elements. That is, a coordinate is zero-padded on the right. When ordered lexicographically, the nodes for each coordinate appear in order according to a depth-first pre-order traversal of the tree. Each coordinate uniquely identifies a single node. Every ancestor’s coordinate is a prefix of any child’s coordinate, ignoring zeros. From the above it follows that every node is lexicographically greater than its left sibling and differs from it by exactly one non-zero element, and that this element is the last non-zero element in the coordinate.

To understand how we construct these coordinates we must consider how we represent the AST.

1. Encoding the AST

We represent the AST as a matrix of 3 columns and *n* rows, one for each node in the tree. The first column contains the inter-node relationships in the form of a depth vector. The second column is a vector of the node types, while the third contains the “value” of the node, such as the name in a variable.

Parsing the examples above gives the following ASTs:

F(f) E(v)  
 │ ┌───────┼──────┐  
 E E P(÷) E  
 ┌─┴─┐ ┌────┼────┐ ┌────┼────┐  
 F A V(a) P(+) V(b) V(c) P(×) V(d)  
 │ │  
 E N(7)   
 ┌────┼────┐  
V(⍵) P(+) V(⍵)

The depth vector for these trees we name Fd and Ed, respectively:

Fd←0 1 2 3 4 4 4 2 2 3  
Ed←0 1 2 2 2 1 1 2 2 2

The node types we call Ft and Et:

Ft←’FEFEVPVAN’  
Et←’EEVPVPEVPV’

And finally, we call the values vectors Fv and Ev:

Fv←’f’ 0 0 0 ‘⍵’ ‘+’ 0 7  
Ev←’v’ 0 ‘a’ ‘+’ ‘b’ ‘÷’ 0 ‘c’ ‘×’ ‘d’

We combine these to form the respective AST matrices. We write A,B to catenate arrays A and B along their last axes and ⍪A to turn a vector into a 1-column matrix. Thus, the two tree matrices are given by the following expressions:

Fd,Ft,⍪Fv  
0 F f  
1 E 0  
2 F 0  
3 E 0  
4 V ⍵  
4 P +  
4 V ⍵  
2 A 0  
3 N 7  
 Ed,Et,Ev  
0 E v  
1 E 0  
2 V a  
2 P +  
2 V b  
1 P ÷  
1 E 0  
2 V c  
2 P ×  
2 V d

Note especially that all inter-node information lives in the depth vector, but that this information requires non-local access as described in the previous section to use it. Constructing node coordinates fixes this issue.

1. Constructing Node Coordinates

In building the node coordinates, our goal is to build a matrix where each row is a node coordinate satisfying our previously described requirements. We end up with an *n d* shaped matrix where *n* is the number of rows and *d* the depth of the tree. As noted above, the depth vector contains all the necessary information, but in the wrong form.

We write f⌿A to reduce the first axis of A using function f. Thus, +⌿V is the sum of the elements in vector V. The function x⌈y gives the maximum of its two arguments. So, the depth of each tree is given by the following:

1+⌈⌿Fd  
5  
 1+⌈⌿Ed  
3

We can obtain the ordered sequence by writing ⍳n:

⍳3  
0 1 2

So the depths of all nodes that appear in the depth vectors is thus:

⍳1+⌈⌿Fd  
0 1 2 3 4 5  
 ⍳1+⌈⌿Ed  
0 1 2

|  |  |  |
| --- | --- | --- |
|  | 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) |
| ↓ | Drop left elements from right | Matrix to Vector of Vectors |
| ∧ | Boolean AND | N/A |
| ∨ | Boolean OR | N/A |
| ⊢ | Right argument | Identity |
| ⊣ | Left argument | Identity |
| ⍉ | N/A | Transpose |

**Table 1.**Summary of Functions

The function table or outer product of f over vectors U and V is written U ∘.f V giving a U V shaped matrix as a result. Thus, (⍳3)∘.×⍳3 gives a small multiplication table:

(⍳3)∘.×⍳3  
0 0 0  
0 1 2  
0 2 4

If we use = instead, we have a Boolean identity matrix:

(⍳3)∘.=⍳3  
1 0 0  
0 1 0  
0 0 1

If we use ∘.= on the depth vector and its set of depths instead, we see an expanded Boolean representation of the depth vector:

Fd∘.=⍳1+⌈⌿Fd  
1 0 0 0 0  
0 1 0 0 0  
0 0 1 0 0  
0 0 0 1 0  
0 0 0 0 1  
0 0 0 0 1  
0 0 0 0 1  
0 0 1 0 0  
0 0 0 1 0  
 Ed∘.=⍳1+⌈⌿Ed  
1 0 0  
0 1 0  
0 0 1  
0 0 1  
0 0 1  
0 1 0  
0 1 0  
0 0 1  
0 0 1  
0 0 1

These matrices let us see the nesting features of each tree more visually, but also suggest another step. We can compute a sum scan with +⍀, also called a prefix sum, along the first axis. Applying this function on the above matrices leads to an interesting result:

+⍀Fd∘.=⍳1+⌈⌿Fd  
1 0 0 0 0  
1 1 0 0 0  
1 1 1 0 0  
1 1 1 1 0  
1 1 1 1 1  
1 1 1 1 2  
1 1 1 1 3  
1 1 2 1 3  
1 1 2 2 3  
 +⍀Ed∘.=⍳1+⌈⌿Ed  
1 0 0  
1 1 0  
1 1 1  
1 1 2  
1 1 3  
1 2 3  
1 3 3  
1 3 4  
1 3 5  
1 3 6

These matrices are lexicographically ordered, and each ancestor shares a common prefix with its descendants. They are also unique coordinates. Only the spurious digits at the end of each coordinate prevent these matrices from meeting all our requirements for valid node coordinates.

The expression V f⍤¯1⊢M applies f to corresponding elements of V and rows of M:

3 3⍴⍳9  
0 1 2  
3 4 5  
6 7 8  
 (⍳3)+⍤¯1⊢3 3⍴⍳9  
0 1 2  
4 5 6  
8 9 10

If n↑V takes the first n elements of V, then we can obtain coordinate matrices from the prefix sums by noting that the spurious digits all come after column d+1 where d is the depth corresponding to that coordinate. The following gives a complete expression for computing a node coordinate matrix from a depth vector, shown using Fd and Fe:

|  |  |
| --- | --- |
|  | 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 |

**Table 2.** Summary of Operators

⊢Fc←(1+Fd)↑⍤¯1⊢+⍀Fd∘.=⍳1+⌈⌿Fd  
1 0 0 0 0  
1 1 0 0 0  
1 1 1 0 0  
1 1 1 1 0  
1 1 1 1 1  
1 1 1 1 2  
1 1 1 1 3  
1 1 2 0 0  
1 1 2 2 0  
 ⊢Ec←(1+Ed)↑⍤¯1⊢+⍀Ed∘.=⍳1+⌈⌿Ed  
1 0 0  
1 1 0  
1 1 1  
1 1 2  
1 1 3  
1 2 0  
1 3 0  
1 3 4  
1 3 5  
1 3 6

A careful study of the definition of a node coordinate and the above construction should reveal why this works. Intuitively, we are creating a multi-dimensional space or a number system in which each digit place or dimension contains or circumscribes a smaller space in which are contained all the descendant nodes that appear lower in the tree. Each coordinate is a sort of special path through the tree encoded to have desirable properties relative to other paths.

1. Operations on Node Coordinates

The simplest operation over a node coordinate is to extract the depth of the node. The expression C⍳0 finds the first occurrence of 0 in C and returns the index of that occurrence. (The application of ⍳ here is its dyadic form, meaning that it receives two arguments. Previous uses of ⍳ have used its monadic form, which generates all indices, instead of finding a specific index. Most APL functions have a dyadic and monadic form.) Thus, the depth of a node is given by the following:

C←1 1 2 0 0  
 ¯1+C⍳0  
2

The primary calculation we care about in order to do function lifting, expression flattening, and other sorts of flattening transformations is to determine which nodes are ancestors of another node. The core of this computation determines whether one node is a child of another based on their node coordinates. This amounts to computing whether one node is a prefix of another, ignoring zeros.

We write (f g h) to represent the composition of functions f, g, and h as a *function train*. This is used with another shorthand of composition according to the following equivalences:

A(f g h)B ←→ (A f B) g (A h B)  
A(0 f h)B ←→ 0 f (A h B) ⍝ Any constant for 0  
A(f g)B ←→ f (A g B)

With this, we can write a function to compute whether a given coordinate is a prefix of another.

P←1 1 0 0 0  
 C(=∨0=⊢)P  
1 1 1 1 1  
 ∧⌿C(=∨0=⊢)P  
1

Here P is the coordinate we wish to check against C, to determine whether C is a descendant. The functions are all standard logical functions extended to apply point-wise over arrays. The function ⊢ always returns its right argument. The function (=∨0=⊢) reads intuitively as, “equal or right element is zero.” In this case to make this a predicate we combine this with the “for all” reduction using ∧⌿. This particular pattern is a special case of inner product, which we can compute using f.g. Thus, +.× is a function that computes traditional matrix-matrix multiplication when applied to two matrices. However, we can also use it to compute the above reduction and application:

C∧.(=∨0=⊢)P  
1

This extends the reduction to allow us to use matrix values for C and P instead of vectors, and thus determine whether rows in C are descendants of multiple candidates given by P. This becomes important to using the prefix function. In our examples, we want to determine the nearest ancestor of a certain node type to which a node belongs. In the case of function lifting, we want to determine the nearest ancestor function, in the case of expression flattening, we want to determine the nearest ancestor expression node. We can extract the rows we care about for each example using ⌷, where i⌷M returns the *i*-th row of M. In the following expressions, we use 0 2 rather than ¯1 in ⍤ to indicate that ⌷ will receive the entire matrix on the right for each scalar index on the left.

⊢Fp←0 2⌷⍤0 2⊢Fc  
1 0 0 0 0  
1 1 1 0 0  
 ⊢Ep←0 1 6⌷⍤0 2⊢Ec  
1 0 0  
1 1 0  
1 3 0

The matrices Fp and Ep correspond to the function and expression node coordinates in our two examples, respectively. We could now use Fp and Ep to compare against Fc and Ec and determine which nodes contained which nodes in the tree. However, our prefix function will return 1 when we ask whether a node is a prefix of itself. In this case we don’t want that. The solution to this is to drop the last non-zero element from the coordinate. This will not prevent it from matching against any of its ancestors, but will prevent matching against itself. We can use the depth vector with the ↑ function to take all but the last non-zero element. We need to remember to extend the returned array with an extra zero column, since the shape will be too small otherwise.

⊢Fcp←(Fd↑⍤¯1⊢Fc),0  
0 0 0 0 0  
1 0 0 0 0  
1 1 0 0 0  
1 1 1 0 0  
1 1 1 1 0  
1 1 1 1 0  
1 1 1 1 0  
1 1 0 0 0  
1 1 2 0 0  
 ⊢Ecp←(Ed↑⍤¯1⊢Ec),0  
0 0 0  
1 0 0  
1 1 0  
1 1 0  
1 1 0  
1 0 0  
1 0 0  
1 3 0  
1 3 0  
1 3 0

We can now use Fcp and Ecp to determine which Function and Expression nodes match for each node:

Fcp∧.(=∨0=⊢)⍉Fp  
0 0  
1 0  
1 0  
1 1  
1 1  
1 1  
1 1  
1 0  
1 0  
 Ecp∧.(=∨0=⊢)⍉Ep  
0 0 0  
1 0 0  
1 1 0  
1 1 0  
1 1 0  
1 0 0  
1 0 0  
1 0 1  
1 0 1  
1 0 1

We use the ⍉ function, which computes the transpose of its right argument, to ensure that the row and column sizes match up. With the above, we can determine the “greatest” match, which is the closest. We can replace each 1 in the above matrices by their column numbers (the number of the column in which that 1 occurs), and then we can take the maximum column number in each row to determine an index in either Fp or Ep that is the appropriate “parent” coordinate for that node. This works because we have made sure to order the Ep and Fp matrices lexicographically. The following shows this computation for the expression example; we elide the function example as redundant at this point. We also use ⌈/ instead of ⌈⌿ to reduce along the last, rather than the first, axis.

⊢Ei←⌈/(⍳3)×⍤1⊢Ecp∧.(=∨0=⊢)⍉Ep  
0 0 1 1 1 0 0 2 2 2  
 ⊢Ek←Ei⌷⍤0 2⊢Ep   
1 0 0  
1 0 0  
1 1 0  
1 1 0  
1 1 0  
1 0 0  
1 0 0  
1 3 0  
1 3 0  
1 3 0

At this point we have two values, Ek and Fk, which indicate the closest containing node that we care about for each node in the tree, using its node coordinate.

Fk  
1 0 0 0 0  
1 0 0 0 0  
1 0 0 0 0  
1 1 1 0 0  
1 1 1 0 0  
1 1 1 0 0  
1 1 1 0 0  
1 0 0 0 0  
1 0 0 0 0

We will use these keys to perform computation over the AST and accomplish our tasks of function lifting and expression flattening in the next section, but we make a final note here, that we can imagine many other operations which might be used throughout the compiler when we care about how nodes relate to each other. The use of operators like reduction, scan, inner and outer products allow us to obtain the necessary information from the node coordinate matrix. Sometimes these results might not be obvious, but the example of the “belongs to” relationship given above demonstrates a pattern that arises repeatedly throughout our data parallel compiler, and is therefore particularly useful to understand.

1. Computing with Node Coordinates

Given the above keys and grouping information, we now have all the information we need to do function lifting and expression flattening, but we lack a suitable operator to complete the task. Indeed, it isn’t immediately apparent that a single, general purpose operator over arrays exists which can accomplish such a task. However, the Key operator, while masquerading as a useful business analytics operator, represents just such a useful tree transformation tool.

* 1. The Key Operator

The Key operator (written ⌸), takes a single function that expects two arguments and returns a function which takes two arguments. The left argument is a set of keys, and the right argument is a set of corresponding elements associated with those keys. In our case, we provide matrices for both of these arguments, so each pair of rows corresponds to a key-value pair. A simple example of the key operation at work is to compute a histogram (≢ here can be thought of as “tally” or count):

⊢X←?10⍴5  
1 1 4 0 1 3 1 1 2 1  
 X(⊣,(≢⊢))⌸X  
1 6  
4 1  
0 1  
3 1  
2 1

To understand a bit better how the Key operator applies its function, consider the function {⍺ ⍵} which returns the pair of its right and left arguments. If we apply it to the same value as above, we get the following:

X{⍺ ⍵}⌸X  
1 1 1 1 1 1 1   
4 4   
0 0   
3 3   
2 2

In our case, we use either Fk or Ek as our keys applied to the corresponding AST. We also will drop off the first row in each AST using 1↓ since this node “contains” everything. In a complete AST this is usually the Module boundary node which contains the entire set of functions and values in the module.

* 1. Function Lifting

If we use the key operator with Fk, we get the following (the monadic use of ↓ converts Fc from a matrix to a vector of vectors):

(1↓Fk){⍺ ⍵}⌸1↓Fd,Ft,Fv,⍪↓Fc  
┌─────────┬─────────────────┐  
│1 0 0 0 0│┌─┬─┬─┬─────────┐│  
│ ││1│E│0│1 1 0 0 0││  
│ │├─┼─┼─┼─────────┤│  
│ ││2│F│0│1 1 1 0 0││  
│ │├─┼─┼─┼─────────┤│  
│ ││2│A│0│1 1 2 0 0││  
│ │├─┼─┼─┼─────────┤│  
│ ││3│N│7│1 1 2 2 0││  
│ │└─┴─┴─┴─────────┘│  
├─────────┼─────────────────┤  
│1 1 1 0 0│┌─┬─┬─┬─────────┐│  
│ ││3│E│0│1 1 1 1 0││  
│ │├─┼─┼─┼─────────┤│  
│ ││4│V│⍵│1 1 1 1 1││  
│ │├─┼─┼─┼─────────┤│  
│ ││4│P│+│1 1 1 1 2││  
│ │├─┼─┼─┼─────────┤│  
│ ││4│V│⍵│1 1 1 1 3││  
│ │└─┴─┴─┴─────────┘│  
└─────────┴─────────────────┘

Notice that we have now grouped all of the relevant parts of the tree according to which nodes would appear in their respective functions after lifting. Refer to the original lifting example in the introduction to verify this. Indeed. The second row in the above example shows the internal function complete and ready to name. Each element in the second column corresponds to the body of one of our lifted functions. In the case of the first function, the outer function, we have a spurious function node in the body. This is intentional. When we lift these functions, we will replace each spurious function node with a variable node referring to the function’s generated name.

Each of these function bodies has a specific coordinate associated with it. Because these coordinates are uniquely identifying, we can use these as input into a name generator to generate names that we know are unique for each function body. Furthermore, because we retain this information in the corresponding function nodes that appear in the body of each function to be lifted, we know exactly what name that function has been given, and we can replace the function node with a variable node referencing that name instead, without referring to any state outside of the immediate information given to the function lifter. Indeed, each row in the above matrix represents a function lifting task that can be completed without any additional information. That is, there are no dependencies between rows to perform lifting. This gives us a straightforward parallel execution of function lifting.

The final task to complete function lifting of each function body before lifting is to shift the depths in the depth vectors to correspond to those of a function lifted to the top-level, and to attach a function node to the top of each of the bodies. At that point, we simply recombine all of the newly created functions into a single top level.

* 1. Expression Flattening

If we use Ek as the key for the expression example, we get the following:

(1↓Ek){⍺ ⍵}⌸1↓Ed,Et,Ev,⍪↓Ec  
┌─────┬─────────────┐  
│1 0 0│┌─┬─┬─┬─────┐│  
│ ││1│E│0│1 1 0││  
│ │├─┼─┼─┼─────┤│  
│ ││1│P│÷│1 2 0││  
│ │├─┼─┼─┼─────┤│  
│ ││1│E│0│1 3 0││  
│ │└─┴─┴─┴─────┘│  
├─────┼─────────────┤  
│1 1 0│┌─┬─┬─┬─────┐│  
│ ││2│V│a│1 1 1││  
│ │├─┼─┼─┼─────┤│  
│ ││2│P│+│1 1 2││  
│ │├─┼─┼─┼─────┤│  
│ ││2│V│b│1 1 3││  
│ │└─┴─┴─┴─────┘│  
├─────┼─────────────┤  
│1 3 0│┌─┬─┬─┬─────┐│  
│ ││2│V│c│1 3 4││  
│ │├─┼─┼─┼─────┤│  
│ ││2│P│×│1 3 5││  
│ │├─┼─┼─┼─────┤│  
│ ││2│V│d│1 3 6││  
│ │└─┴─┴─┴─────┘│  
└─────┴─────────────┘

Again, we can see immediately that we have grouped each set of nodes according to the expressions that are to be lifted. Just as in the case of function lifting, we can adjust the depths of each expression to the correct depth and we can replace each expression node with a variable reference based on that node’s coordinate. Each expression can be given a unique name based on its coordinate. A later compiler pass can reduce these names down to the minimum actually required to represent the expression if desired.

The only extra issue involved here is to ensure that the order of evaluation matches. In our case, we are assuming that the order of evaluation is right to left, which means that the above order is actually backwards of the desired order. During recombination, we simply reverse these orders and this fixes that problem. More work would be required to take into consideration a specific precedence hierarchy.

1. Scaling up, Considerations and Thoughts

How well do these techniques work in practice? We have applied these techniques to an entire compiler, with the same restrictions as used here. The result was a success, and the core of the compiler, which is everything but parsing and code generation, is contained in a single file consisting of only 40 lines. In 12 point font, the compiler easily fits on a single sheet of paper. It produces C and CUDA code.

Not every pass in the compiler requires this sort of operation. Indeed, if flattening occurs early and in the correct order, it’s possible to use much simpler methods of grouping using only a prefix sum scan instead of operating over the node coordinates. It’s also interesting to note that we do not need to recompute the node coordinates throughout the compiler. Indeed, this would defeat their purpose somewhat, as the node coordinates allow us to understand the original position of a node, even if that node has now been lifted elsewhere. We use this information when resolving lexical scope, for instance.

When writing data-parallel code, duplicated work is common. The existing compiler contains little to no inherently duplicated work. Most of the duplicated work exists for brevity and clarity of the code, rather than for any required reason, and could easily be factored out.

Finally, it is interesting to note that the simplified control flow gives rise to some positive side-effects. Besides brevity and directness, the complexity of each operation and the composition of operations is well known. We have not yet tried to do this, but it is straightforward to derive a worst-case complexity bound on code written in this style, so simple, in fact, that it can be accomplished automatically. It is conceivable then, that the compiler could produce not only an executable program from code written in this style, but also a complexity bound on its execution.

1. Related Work

The J programming language [18] 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 [1, 17].

Fritz Henglein demonstrated a class of operations, called discriminators*,* of which the Key operator is a member. [16] Namely, a discriminator performs the same grouping computation as Key, but does not apply a function over these groups with their keys. Henglein provides a linear implementation of these operations.

The EigenCFA effort [22] 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 [20] 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 [2] developed vectorized approaches to handling certain analyses to compile traditional APL, including a SIMD tokenizer [4]. 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 [3].

Timothy Budd implemented a compiler [5, 6] 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 [21].

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. [8, 9, 10, 11, 13, 14, 15]

J. D. Bunda and J. A. Gerth presented a method for doing table driven parsing of APL which suggested a parallel optimization for parsing, but did not elucidate the algorithm [7].

1. Conclusion

We have derived a method of performing computation over sub-trees selected on the basis of inter-node properties through the use of the Key (⌸) operator and node coordinates, which enable local computation of these inter-node properties. This method is both general and direct, and when combined with traditional and more mundane array programming, suffices to implement the complete core of a compiler, modulo parsing and code generation. The method requires no special operations or unique special casing primitives in the language. Moreover, it is strictly data-parallel and data-flow, without any complex control flow, which not only results in concise code, but also enables much easier analysis of properties such as runtime complexity and verification of correctness.

We have demonstrated the technique and the core insights behind the data structures involved. It presents a solution to a very old and traditional problem in a very uncommon light, by eschewing the common practices that underlie every other significant and general solution found in modern compilers today and replacing them with an entirely different paradigm centred on parallelism and aggregate operations.

This research is supported by Dyalog, Ltd. to develop a high-performance compiler for APL.

[1] 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

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

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

[4] 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.

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

[6] 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

[7] 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

[8] 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

[9] 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.

[10] 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

[11] 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

[13] 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

[14] 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

[15] 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>

[16] Fritz Henglein. 2008. Generic discrimination: sorting and paritioning unshared data in linear time. In Proceedings of the 13th ACM SIGPLAN international conference on Functional programming (ICFP '08). ACM, New York, NY, USA, 91-102. DOI=10.1145/1411204.1411220 http://doi.acm.org/10.1145/1411204.1411220

[17] 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

[18] Roger Hui. 2014. Essays/Key. <http://www.jsoftware.com/jwiki/Essays/Key>

[19] Andrew W. Keep and R. Kent Dybvig. 2013. A nanopass framework for commercial compiler development. In *Proceedings of the 18th ACM SIGPLAN international conference on Functional programming* (ICFP '13). ACM, New York, NY, USA, 343-350. DOI=10.1145/2500365.2500618 http://doi.acm.org/10.1145/2500365.2500618

[20] 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

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

[22] 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/10.1145/1926385.192644