A DATA PARALLEL COMPILER HOSTED ON THE GPU

Aaron Wen-yao Hsu

Submitted to the faculty of the University Graduate School
in partial fulfillment of the requirements
for the degree
Doctor of Philosophy
in the School of Informatics, Computing, and Engineering,
Indiana University
November 2019

Accepted by the Graduate Faculty, Indiana University, in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Doctoral Committee

Andrew Lumsdaine, Ph.D.

Ryan Newton, Ph.D.

Amr Sabry, Ph.D.

Arun Chauhan, Ph.D.

Joshua Danish, Ph.D.

October 10th, 2019

To my wife Johanna and my daughters Mikaela and Saraqael, who have known no other life than that of aid and support to a seemingly lifelong student, but nonetheless endure faithfully and patiently, and to the family they help me to remain a part of. May God bless you and keep you, and may He grant you many years.

Acknowledgements

I must highlight the patience and endurance of my doctoral committee who, perhaps despite their better judgment, gave me the space and time necessary to pursue an idea that bore none of the traditional markers of good thesis work. Only their willingness to permit my exploration and discovery enabled me to engage with these ideas within my formal education in a way that would never have been possible without. I strongly suspect that they may not have known whether the rope they were giving me would ultimately serve to aid my academic climb or strangle me where I stood.

Likewise, the pursuit of this topic would not have been possible without the generous, tireless, and optimistic aid provided by Dyalog, Ltd. Their passion for the future of APL, for the cultivation of computing that empowers the individual mind, and their investment in young and oftentimes brash lone wolves like myself not only supported this work, but also gave spark to its shape and color, in a way that would not be possible without the exposure they enabled to great minds such as Morten Kromberg, Gitte Christensen, Roger Hui, Stephen Taylor, John Scholes, Robert Bernecky, and the whole of the extended Dyalog APL programming community.

Finally, inordinate recognition must go to Kent Dybvig, without whom this journey would have ended long before it even began.

Aaron Wen-yao Hsu

A DATA PARALLEL COMPILER HOSTED ON THE GPU

This work describes a general, scalable method for building data-parallel by construction tree transformations that exhibit simplicity, directness of expression, and high-performance on both CPU and GPU architectures when executed on either interpreted or compiled platforms across a wide range of data sizes, as exemplified and expounded by the exposition of a complete compiler for a lexically scoped, functionally oriented programming commercial language. The entire source code to the compiler written in this method requires only 17 lines of simple code compared to roughly 1000 lines of equivalent code in the domain-specific compiler construction framework, Nanopass, and requires no domain specific techniques, libraries, or infrastructure support. It requires no sophisticated abstraction barriers to retain its concision and simplicity of form. The execution performance of the compiler scales along multiple dimensions: it consistently outperforms the equivalent traditional compiler by orders of magnitude in memory usage and run time at all data sizes and achieves this performance on both interpreted and compiled platforms across CPU and GPU hardware using a single source code for both architectures and no hardware-specific annotations or code. It does not use any novel domain-specific inventions of technique or process, nor does it use any sophisticated language or platform support. Indeed, the source does not utilize branching, conditionals, if statements, pattern matching, ADTs, recursions, explicit looping, or other non-trivial control or dispatch, nor any specialized data models.

Andrew Lumsdaine, Ph.D.

Ryan Newton, Ph.D.

Amr Sabry, Ph.D.

Arun Chauhan, Ph.D.

Joshua Danish, Ph.D.

Table of Contents

List of Figures

List of Tables

1. INTRODUCTION

Parallel hardware and memory bandwidth constraints dominate current computing architectures. They exhibit high degrees of parallelism with CPU and GPGPU designs that utilize ever greater vector processing units to increase computing power. This increases computational performance on compute-bound problems that can be parallelized. Often, on modern computing hardware, data throughput constrains the potential performance of systems more than sheer processing power.

This presents an interesting challenge for tree transformations. Programmers often use tree or hierarchical data structures, such as ADTs and linked structures to model a wide variety of problems and information. Working with these structures usually involves traversing over the structure and sometimes transforming it. Traversal algorithms visit the nodes of the tree without changing them or modifying their inter-node structure, though they may annotate them with additional information. Many traditional graph problems in high-performance computing are traversals, such as the Graph 500 benchmark (Murphy et al. 2010). Transformations usually involve some sort of traversal, but they primarily modify the structure of the tree. They abound in modern computing, in domains such as DOM processing for web applications (Hors et al. 2004), document processing and print publishing (Saxonica 2017), query optimization (Chaudhuri 1998), and compilers (Keep and Dybvig 2013).

Often theoretically parallelizable, tree transformations subvert casual parallel implementation (Goldfarb, Jo, and Kulkarni 2013) for several reasons. Often the trees exhibit irregular structure, so that despite the apparent parallelism available, extracting and utilizing that parallelism within the tree structure to any great effect proves difficult. Functional programming provides a conceptually parallel

framework for transformations, but in practice, the actual traversal and transformation algorithms used contain more stateful operations than may at first appear (Marlow and Jones 2012), resulting in difficult ordering dependencies that hamper parallel execution. Additionally, programmers almost universally express tree transformations as some form of structural recursion over the tree, with the tree's branching, and therefore, potential available parallelism, increasing as the operation continues, necessitating some sort of dynamic thread spawning in order to take advantage of additional parallelism as it arrives in the recursive algorithm. Finally, the near ubiquitous reliance on recursive algorithms as the foundational algorithmic technique within the field of tree transformations presents challenges for execution on SIMD parallel vector machines (Goldfarb, Jo, and Kulkarni 2013), available on most modern CPUs and GPUs, whose use facilitates peak performance on these architectures.

Other domains have techniques for addressing irregular data patterns, dynamic parallelism, and recursive structures, but the usability of these approaches generally evokes the traditional view that parallel programming is particularly hard, and that parallel programs in general are inherently more complicated than single threaded models. Thus, programmers commonly question not only the theoretical feasibility of parallelization, but also whether the performance gained warrants the presumed increase in complexity, overhead, and programming challenges that will presumably arise because of parallelization (McKenney 2017).

Programming language compilers present a tidy suite of specific tree transformation algorithms and offer an interesting, self-contained domain by virtue of their long history, relatively

uniform approach to representation and description, and strong reliance on transformations as the foundational operation. The trees over which they compute generally exhibit high degrees of irregularity and traditional approaches nearly all describe the tree transformations (called passes) over the AST (abstract syntax tree) or IR (intermediate representation) using some form of structural recursion over the tree, even in cases where some of this is abstracted or hidden away. Functional programming has proven highly successful for working with compilers (Keep 2013), but even within functional programming communities, the challenges to parallelizing the compilation process has proven difficult enough that even if theoretically possible, no one has deployed a scalable method practice (Marlow and Jones 2012). Indeed, when it comes to successful parallelization of compiler transformations on fine-grained, data parallel hardware (such as SIMD CPU operations or GPUs), failure is not just the rule, but nearly the axiom. Some teams have developed ad hoc, specialized implementations of good performance on some specific compiler analyses targeting GPU execution and fine-grained parallelism (Mendez-Lojo, Burtscher, and Pingali 2012; Prabhu et al. 2011), but these passes not only required significant human effort and technical engineering, they are also primarily traversals and not transformations.

In general, the core compiler transformations contain a relative dearth of computation and a high degree of permutation or manipulation of data in irregular patterns, making traditional algorithms exceptionally difficult to parallelize for non-obvious performance gains. Such transformations not only exhibit the classical issues of irregularity and recursion, thus implying some form of dynamic parallelism is required, but also fail to meet the “return on investment” criteria

discussed above. Namely, because the transformations are so often memory-bound rather than compute-bound, dynamically scheduled parallelism techniques (Cong et al. 2008) can exhibit high overheads and significant increases in programming complexity that must be carefully managed to achieve performance, and the cost of learning and using such systems can discourage their use as de facto standards for implementing such algorithms; they are used, instead, only when the problem size or performance penalties of more standard approaches suggests that the use of such additional layers is worth it. The increased programmer and computing overheads, higher risk for error, and increased source code burden compels such techniques into use only when performance considerations warrant the potential increased costs, limiting the general application of these techniques to a few, select cases, if used at all. They certainly fail to demonstrate sufficient usability to recommend them as standard techniques to subsume or replace the traditional approaches in constructing compilers. In other words, coarse-grained parallel solutions centering around task-parallelism do not appear to have the necessary scalability of parallelism to see significant benefits for their increased complexity of use when applied within a compiler even if they could be made to work, of which there is scant evidence for their successful widespread application across the range of compiler transformations.

While the author has been unable to source a single successful attempt at general parallelization of the compiler transformations themselves, parallel build systems have proven highly successful and have represented the usual “unit of granularity” by which compilers have been parallelized. Much of the work within parallelizing the build process has been in decoupling individual units of source code (often called modules) so that each unit may be compiled by a separate

compilation process. This allows for a parallelizing build system to execute multiple compilation operations in parallel on distributed or SMP computing environments. While this works well, it leaves the available computational power available through SIMD architectures untapped and inaccessible to a wide range of algorithms.

Work on accelerating particularly expensive elements of compiler design, such as the EigenCFA algorithms (Prabhu et al. 2011), is limited by the need to transfer data back and forth from the GPU. This work addresses the missing "glue" components that take the form of tree transformations, enabling both high-performance standalone tree transformations, but also reducing the need to transfer data back and forth from GPU to CPU, reducing the critical paths for the total application architectures and improving the potential performance scaling of existing GPU algorithms.

This work proposes a novel approach to compiler construction that addresses not only the parallel execution of compiler transformations on GPUs and vector machines but also the related issues of generality and complexity of the programming model. It provides methodology of compiler construction for fine-grained, data parallel hardware that is suitably general, performant, and simple that it may be applied en masse, without requiring the maintenance of a separate, single-threaded version of the source code. To accomplish this, the method alters the foundational assumptions of how to represent both the data and the computation around tree transformations. Specifically, it avoids recursion, branching, and any other operation that is not inherently or readily parallelized. It also avoids the use of records (ADTs) as the primary method of data abstraction. Instead, it builds compiler

transformations entirely through the composition of data parallel operators over *n*-dimensional, rectilinear arrays using APL as the programming model and notation. It additionally avoids the use of specialized data abstractions for trees and avoids abstractions that might obscure the source code's directness of expression. Instead, it uses traditional array programming idioms directly as tree transformation idioms.

This work demonstrates the method by constructing a compiler over a lexically scoped, dynamically typed APL syntax known as Co-dfns, which contains all the traditional constructs of recursive, functional-style programming, such as Scheme, including IF-statements (branching) and recursion, in a simplified form. The input language to the compiler includes higher-order programming but does not include first-class procedures. The syntax itself is a production language widely used and found in the Dyalog APL interpreter. The compiler itself exhibits high performance on both CPU and GPU architectures, executing on both compiled and interpreted implementations, and outperforms equivalent compilers implemented using more traditional methods of structural recursion written in Nanopass. The performance scales both to large source sizes as well as small source sizes and is completely agnostic of the unit of granularity. That is, it can compile a single small piece of source in parallel as well as compile any number of independent source modules in parallel without the requirement for a separate build system to manage parallel dispatch of separate instances of the compiler; the same interface for compilation works for single modules and multiple modules, all of which are parallel by default without loss of performance at small data sizes.

Furthermore, when compared against equivalent compilers implemented in the Nanopass framework, which is a framework specifically designed for efficient and convenient compiler pass design in a functional style on Scheme-like platforms, the compiler is significantly shorter, smaller, and simpler in design and semantics, requiring no additional tooling, runtime support, or specialized libraries or syntaxes, and yet it retains significant advantages in size and complexity versus the domain-specific Nanopass framework.

This thesis makes the following contributions:

- The compiler demonstrates the sufficient power, expressivity, and usability of traditional array programming models for arbitrary tree manipulation computations traditionally viewed as difficult or intractable to parallelize efficiently and productively onto vector architectures.
- The compiler exhibits runtime and memory performance improvements over existing methods on both CPU and GPU vector architectures.
- The techniques fill a gap in compiler design, enabling more expensive analysis algorithms to run on the GPU without needing to shuffle data back and forth from the GPU to the CPU in-between more expensive analyses, because the tree transformation elements can now be executed efficiently on the GPU as well.
- The compiler scales to a wide range of data sizes across both CPU and GPU architectures.
- The source code of the compiler is significantly shorter and simpler in structure and semantics than existing methods.

- The asymptotic complexity of the compiler is entirely within the computational capabilities of normal mechanical proof systems, requiring little to no human proof expertise to calculate; this includes near byte-level accuracy in calculating memory requirements for AST storage and use as well as memory access patterns.
- The compiler requires no new domain specific techniques, language features, or programming abstractions, and does not require sophisticated compiler or toolchain support to achieve high performance.
- All programming techniques and primitives used are general purpose and transfer from or to both other tree manipulation algorithms as well as other programming domains; thus, the methods used require no invention either in structure, method, or approach over traditional array programming concepts that are in widespread use.

2. BACKGROUND

2.1. APL vis a vis Iverson-style Array Programming

Kenneth Iverson invented and refined the APL notation from the 1950's through the 1960's (Falkoff 1978). Originally called "Iverson Notation," it eventually became "A Programming Language" after Iverson's seminal work of the same name (1962). As envisioned, Iverson and the community that grew up and built the APL language always insisted that the language itself was, at its essence, a notation designed to replace mathematics and to enhance the user's ability to work with complex ideas (Iverson 2007; McIntyre 1991). Knuth emphasized this idea when he called APL a language for people who "want a nice elegant way to state the solution to their problem..." (1993).

In the author's opinion, it remains one of the most consistent and generally scalable notations for describing complex algorithms in a concise, efficient manner, without excessive detail that might obscure the core computation. In this work, it is used as the programming language, mathematical notation, and general working notation for all precise discussion of the computations involved. It is uniquely well-suited for this task as the most mature, commercially widespread, and well-studied notations for such work, having been deployed in the millions of lines of code across many decades of mission critical work (Hallenberg 2016). Its behavior under multiple parallel systems has been studied, its performance model is well established (Bernecky 1997; Bernecky and Scholz 2015; Budd 1984; Ching 1990; Hsu 2014, 2015; Ching and Katz 1994; Ju and Ching 1991; Ju, Ching, and Wu 1991;

Schwarz 1991), and significant work has been done on its semantic foundation (Mullin 1988; Slepak 2014).

Perhaps more importantly, however, this work emphasizes the use of well-established, historically anchored computational ideas. APL is among the oldest and longest established continuously deployed commercial programming languages, most particularly in the domain of parallel vector machines (Falkoff 1978). This work emphasizes the construction of high-performance tree manipulation on data parallel hardware without the introduction of any fundamentally new foundational programming models, syntax, or semantics. The use of APL in its traditional forms serves to reinforce this idea, and its models are easily transferrable to any number of other programming languages, as well as representing an easy translation into other mathematical notations.

However, data parallel programming of the APL sort is not well understood by many programmers. In particular, Compiler requires a careful understanding of the nature of APL, its data model, syntax, semantics, and performance model, as this section presumes an understanding of APL notation and its relation to the performance model to avoid an explosive growth in that section's content. To address this, this section details the APL notation as it is used in future sections. A quick reference to the APL primitives is also included as an appendix to facilitate reading Section 3.

To properly understand APL, the syntax, its data model, the classification of its primitives, its idiomatic use, and its fundamental performance model are all important, and will be dealt with in this section individually.

2.1.1. Array Model

APL's model of arrays is specific and somewhat more general than often found in other programming languages. APL (absent any extensions for OOP or other forms of data modeling) models all data as an array of some sort. All arrays have a uniform structure consisting of two parts, the shape and the values. An array is an object whose set of values are ordered and arranged according to its shape. In APL, all shapes are rectilinear and all values are discreet objects (also arrays). This means that all shapes may be represented by an ordered set of non-negative integers.

A shape describes a (possibly empty) rectilinear space in which discreet values may be arranged. A shape with no dimensions is a dot, with a single value. A shape with one dimension is a line; with two, a rectangle; with three, a cube; and with four or more, a higher-dimensioned rectilinear space. Shapes of 0, 1, 2, 3, and more dimensions are called, respectively, scalars, vectors, matrices, cubes, and noble arrays. A shape is itself considered a value (array) in APL, an array of 1 dimension whose values are the dimensions of the shape. Conceptually, a shape describes a set of discreet points in a multidimensional space that serve to arrange values in row-major order, with the first dimension of the shape the most significant. The rank of an array is the length of its shape, that is, the number of dimensions in the shape.

Values in an array are just arrays. As an ordered set of values, a combination of the values listed sequentially as a vector, combined with a shape, suffices to completely describe any given array. The APL model of arrays is known as the "floating" model, because the most atomic units of data, the character and the number, are also considered to be arrays. Characters and numbers in APL are scalar

arrays (that is, arrays of rank 0, having no dimensions) whose value is itself. There are as many such arrays in APL as there are characters and numbers representable in the APL implementation.

Each point within a space described by a shape has a specific index that uniquely refers to that point, allowing any value within an array to be referenced specifically. An index that refers to a single value in an array is a vector of the same length as the array's shape, whose values are integers in the range of `[0,n)`, where `n` is the value of the dimension in the corresponding position of the array's shape. Sub-arrays may be indexed or referred to as well, by leaving some elements or values of an index blank. This blank then refers to the entire range of possible values for that dimension. This allows for an array to be "cut" into various sub-arrays of varying shapes. If these cuts occur only to the most significant elements of the shape (for example, selecting only the first row of a matrix, or the first row of the third matrix of a cube), then the sub-arrays are considered cells of the array, with the major cells of an array consisting of all the trailing dimensions of a shape treated as blank except for the first, or major, one. Thus, a matrix consists of a set of vector major-cells, one for each row of the matrix.

The above definition leads to another interesting concept, depth. The shape of an array describes how values are arranged within a given space described by that shape, but each of those values may themselves have their own shape and values, and so forth down recursively. The concept of depth captures this "other dimensionality." It is also worth noting that a scalar value may be a single numeric or character value, but it may also contain any arbitrary complex array as its single value.

We say that a primitive scalar number or character, such as `5` or `'a'`, has a depth of zero, since it has itself as its own value. For all other values, we say that the depth of an array has a magnitude

(absolute value) equal to one more than the greatest depth of any of its values. If all the values have the same depth, then the depth is said to be positive. If the values have different depths, then the depth is considered negative.

Thus, it is important to understand the difference between depth, shape, and rank. Rank describes the number of dimensions in the shape. The shape is a vector that describes the arrangement of an array's values in space. However, when arrays are other than simple numeric scalars or characters, then the depth of the array captures how deeply nested the array is, that is, how many different complex spaces exist and would need to be traversed to examine all of the simple numeric or character values contained in the array. In the case of a depth 0 array, there is no space to traverse, since the value is itself and already found. In the case of a depth 1 array, that is, an array consisting of primitive simple scalars, then there is only the single shape describing a single space to traverse to find all the values contained in the array. Depth 2 indicates that each array value contained by the depth 2 array itself has a depth of 1, and therefore, requires its own traversal to find the primitive values, for each value in the depth 2 array.

For novices, it is common to confuse the concept of shape and depth, so it is worthwhile to reiterate these concepts in various ways. A given array is always made up entirely of shape and values, and these two things are totally sufficient and necessary to describe the array. Depth is an emergent property that describes the nature of the values of the array, and is defined recursively over the values independent of the shapes, by the following recursive definition: *if the array's single value is itself, then*

return a depth of 0; else, compute x, one more than the maximum absolute value of the depths of all values in the array; return x if all values had the same depth, and -x otherwise.

2.1.2. APL Expression Syntax

This work uses the APL expression syntax combined with the Co-dfns syntax for anonymous function declarations and namespaces, described in a future section. Traditional APL function definition syntax is not used, and the dfns syntax developed by the late John Scholes is favored instead (Dyalog 2019a).

The basic expression syntax obeys the following model, absent the dfns model for function specification:

```
Atom    := literal | Index | ( Expression )
Index   := Atom [ Expression ; … ]
Expr    := Atom | App | Binding
App     := Func Expression | Atom Func Expression
Func    := FnAtom | Func oper | Func oper FnAtom | Func [ Expr ]
Binding := Name ← Expression | Name Func ← Expression
Name    := var+ | var [ Expression ; … ]
FnAtom  := primitive | var | ( Func ) | ( FnBind )
FnBind  := var ← Func
```

The most important element to notice from the above is the nature of application precedence. There are two types of applications in APL, function application and operator application. In both cases, applications may be monadic or dyadic, that is, they may apply to one or two arguments. When composing expressions from values and functions, the application precedence is right to left, that is, they are right associative. In the case where functions are being composed together using operators (which are higher-order operations), the association is to the left, that is, operations bind from the left to the right. Additionally, note that in the above model, operators only take functions as arguments,

but in APL, some operators may also take normal values as arguments (called operands). This omission simplifies the above model without removing its essential elements.

APL's literal notation for arrays allows for writing inline vectors and scalars. The following examples demonstrate the APL literal style sufficiently in depth for this treatise:

```
1               ⍝ A scalar 1
1 2 3           ⍝ A vector containing 1 2 3
'abc'           ⍝ A vector containing a, b, c
'b'             ⍝ A scalar containing b
(1 2 3) 'abc'   ⍝ A vector pair containing 1 2 3 and a b c
```

The indexing and picking function primitives support indexing, but there is also a syntax for indexing called bracket notation. An array `A` is indexed by another array `I` by writing `A[I]`. If one wishes to index a matrix `M` by row and column indices `I` and `J`, then one may write `M[I;J]`. Selecting just specific rows, but all the columns, may be written `M[I;]`, and thus, an entire matrix may be returned as `M[;]`. Higher dimensions follow the same pattern.

Binding names and assigning specific values to specific positions in an array are all accomplished through the same ← primitive, read as "gets." Of special importance here, we note that multiple positions and multiple names may be written as a single conceptual operation, as follows:

```
      X←⍳10
      X
0 1 2 3 4 5 6 7 8 9
      X[0 2 4 6]←¯5
      X
¯5 1 ¯5 3 ¯5 5 ¯5 7 8 9
      X Y←(⍳10)(⍳10)
      X
0 1 2 3 4 5 6 7 8 9
      Y
0 1 2 3 4 5 6 7 8 9
```

This begins to illustrate a general point about APL notation and operation. Generally, operations tend to operate over groups or multiple elements at the same time, including, as above, the indexing operations. There is very little to no formal ordering requirements on these operations, meaning that they semantically permit, but do not require, parallel execution.

Finally, the majority of what gives APL its "special flavor" is its set of primitive functions (first order operations) and operators (second-order operations). These are significant enough that they deserve their own section, as follows.

2.1.3. APL Primitives

APL primitives each receive a one-character symbolic representation. They form the functional building blocks of the language and represent the "runtime library" for most users. Each primitive symbol represents up to two operations, usually conceptually related to each other, distinguished by whether the primitive is applied monadically (that is, with one argument) or dyadically (applied with two arguments). Most primitives receive their canonical definition either by their dyadic or monadic use, with the other use case a special case of the canonical one. As an example, the division operation `÷` is defined over two arguments `Y÷X`, but the monadic application `÷X` means the reciprocal of `X`, or `1÷X`.

Primitives fall into two categories, based on whether they are first or second order operations. First order primitives (called functions) receive only values (arrays) as their arguments and return only values as their results. In APL, the second order operations, called operators, may take either

values or functions (first order operations) as their arguments (called operands to distinguish them from function arguments) and always return function values.

Among primitive functions (that is, first order operations), there are generally two types, scalar and mixed. Scalar primitive functions define a core operation over a scalar value and have a uniform behavior for all values of more complex shapes, while mixed primitives can have varied types of return values and traversal patterns.

Scalar primitives have a primary operation defined by a single computation on a scalar character or number, such as addition, multiplication, division, absolute value, equality, and so forth. APL defines a uniform lifting for all scalar primitives applied over arrays of more complex shapes. Monadic scalar primitives serve as the simplest base from which to understand the lifting, followed by the behavior of dyadic primitives receiving arguments of equal shape, and finally dyadic primitives with arguments that utilize scalar extension.

On a scalar primitive applied monadically, the shape of the returned value is the same as the shape of the argument. The returned array's values are the arrays returned by applying the scalar primitive to each value in the argument, maintaining ordering. A scalar primitive's base case is its definition over a simple scalar array (numeric or character), as each scalar primitive is defined to return a specific computation for simple scalar arrays. For example, the Reciprocal monadic primitive ÷ applied to a number, such as `÷5` will, by definition, return the reciprocal of 5 and will return a domain error on non-numeric simple scalar values. On more complex shapes and depths, the

Reciprocal function will apply pointwise to each simple scalar value while the shape and depth of the input array will be retained.

When applying a scalar primitive function dyadically, it receives a left and a right argument. When these arguments are of the same shape, then the function will return a value whose shape is the same shape as that of its arguments, with values that are the result of applying the scalar primitive recursively with values from the corresponding positions of the left and right arguments. Just as in the monadic case, the base case of such primitives is defined over simple scalars.

It is an error to apply a scalar primitive function dyadically with arguments that do not have the same shape, with one exception, known as scalar extension. If one of the shapes of the arguments is scalar while the other is not, then the scalar argument is used repeatedly as an argument for each value in the other argument during the recursive application. Another way of thinking of this is to imagine that the scalar argument is resized to be the same shape as the other argument, where the scalar value is replicated for each position in the newly shaped array's values.

Here are some examples of scalar primitive function application using basic arithmetic operations, beginning with the definition of a simple matrix:

```
      4 5⍴⍳20 ⍝ 4 by 5 matrix of integers
 0  1  2  3  4
 5  6  7  8  9
10 11 12 13 14
15 16 17 18 19
      X←4 5⍴⍳20
      X
 0  1  2  3  4
 5  6  7  8  9
10 11 12 13 14
15 16 17 18 19
```

```
      ÷X
DOMAIN ERROR: Divide by zero
      ÷X
      ^
      1+X
 1  2  3  4  5
 6  7  8  9 10
11 12 13 14 15
16 17 18 19 20
      ÷1+X
1              0.5            0.3333333333  0.25           0.2
0.1666666667  0.1428571429  0.125         0.1111111111  0.1
0.09090909091 0.08333333333 0.07692307692 0.07142857143
0.06666666667
0.0625         0.05882352941 0.05555555556 0.05263157895 0.05
      X-10
¯10 ¯9 ¯8 ¯7 ¯6
 ¯5 ¯4 ¯3 ¯2 ¯1
  0  1  2  3  4
  5  6  7  8  9
      ×X-10 ⍝ Sign of X-10
¯1 ¯1 ¯1 ¯1 ¯1
¯1 ¯1 ¯1 ¯1 ¯1
 0  1  1  1  1
 1  1  1  1  1
      X+X
 0  2  4  6  8
10 12 14 16 18
20 22 24 26 28
30 32 34 36 38
      X-X
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
```

The use of ⍴ in the above code demonstrates a Mixed primitive operation. Mixed primitives are anything other than scalar primitives, and they do not follow the general rules of scalar primitives. They may return arrays of varying shapes, or may always return values of the same shape, and may likewise return values that have nothing to do with the simple values in the input array or may return the values of the input array unchanged, and so on. As an example, consider the ⍴ shape primitive.

When used monadically, it returns the shape of its argument, while if used dyadically, it reshapes the right argument to have the left argument as its shape.

```
      4 5⍴⍳40
 0  1  2  3  4
 5  6  7  8  9
10 11 12 13 14
15 16 17 18 19
      4 5⍴⍳20
 0  1  2  3  4
 5  6  7  8  9
10 11 12 13 14
15 16 17 18 19
      4 5⍴⍳10
0 1 2 3 4
5 6 7 8 9
0 1 2 3 4
5 6 7 8 9
      ⍳10
0 1 2 3 4 5 6 7 8 9
      ⍴⍳10
10
      5 4⍴⍳10
0 1 2 3
4 5 6 7
8 9 0 1
2 3 4 5
6 7 8 9
```

Another example is the Grade up function ⍋ that computes a sorting permutation vector over its right argument. Here is an example of its use:

```
      ⊢X←?5 5⍴10  ⍝ Random 5 5 matrix
1 3 6 1 3
7 3 8 3 3
4 0 3 6 3
2 0 2 9 1
4 7 0 7 3
      ⍋X
0 3 2 4 1
      X[⍋X;]
1 3 6 1 3
2 0 2 9 1
4 0 3 6 3
4 7 0 7 3
7 3 8 3 3
```

Notice that the Grade Up function always returns a vector, regardless of the shape of its input, and its values define a permutation vector, that is, a set of indices based on the number of major cells in the input.

The appendix contains a listing of APL primitives in quick reference form. Examples are given and used extensively throughout this document, and the use of the primitives within the scope of their application to the domain should be clearly inferred with the help of the quick reference in the appendix.

The available traversal and manipulation capabilities of the mixed and scalar primitive functions in APL is surprisingly diverse, but they would lack expressive power in themselves. APL also defines a set of second-order operators, in both monadic and dyadic form. While the primitive functions often have two definitions based on whether the function is applied monadically or dyadically, the primitive operators are not ambivalent, meaning that they cannot be applied both dyadically and monadically. Each operator must be applied either monadically, or dyadically, but not both. However, in some limited cases, the operator's behavior may change slightly based on whether it has received an array or a function as its operand. When an operator is applied to its operands, it returns what is called a derived function, and then function can then, in turn, be applied to suitable arguments to receive an array result.

Operators generally describe the way in which to apply a given operand function or set of operand functions over the input arguments applied to its derived function. For example, the

reduction operator takes a single function and applies it as the reduction operation over a given dimension of an array, as in the following example:

```
      ⍳20
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
      +⌿⍳20
190
      4 5⍴⍳20
 0  1  2  3  4
 5  6  7  8  9
10 11 12 13 14
15 16 17 18 19
      +⌿4 5⍴⍳20
30 34 38 42 46
      ⍴+⌿4 5⍴⍳20
5
      ⍴+/4 5⍴⍳20
4
      +/4 5⍴⍳20
10 35 60 85
```

Notice that there are two primitive symbols used for reduction, / and ⌿, which operate on the trailing and leading axis (dimension) of the array, respectively. In this case, the reduction eliminates one of the dimensions from the input array in its results, replacing the values with the appropriate reduction along that axis. Additionally, notice that, unlike in function application, a monadically applied operator receives its single operand on the left of the operator primitive, rather than on the right, as would happen with function application, in this case, with the addition function.

As with primitive functions, the appendix includes a quick reference for the APL primitive operators, and it is recommended to make frequent reference to it in the following sections.

For the purposes of our use, it is important to note that APL primitives by design emphasize operating over many elements at a time. Each of them, with very few exceptions, tends to be defined in a data parallel fashion. This means that their order of operations is defined in such a way that it is

often trivially easy to execute them in parallel, especially when composed in a spiritually functional style. For example, when used idiomatically with primitive associative scalar operations, the reduction operation can be implemented using widely implemented reduction techniques on vector machines. The same goes for Scan (Harris 2007), Inner and Outer Product (Silberstein 2008), Key (Sengupta 2007; Satish 2009), and other operators. Additionally, all scalar primitive functions are data parallel by definition and most mixed primitive functions in APL have corresponding data-parallel implementations that have been studied and implemented for GPU and CPU systems, as discussed in a latter section on the overall performance characteristics of APL. However, in order to understand the performance of APL in practice, it is first necessary to spend some time understanding the concept of idiomatic APL.

2.1.4. Idiomatic APL

Technically, APL can be used to write source code that looks very much like the source code seen in any other programming language. Doing so subjects APL interpreters and compilers to the same difficulties as traditional languages in terms of performance and readability. Instead, the APL community developed around the concept of "idiomatic APL" (Eisenberg 1987) that encompass various stylistic ideals about how to construct APL expressions and how to build APL software (Perlis 1977). Of course, ideals are just that, and in practice the degree to which any given APL program expresses these idiomatic principles varies along a wide continuum.

Idiomatic APL tends to be easier to machine analyze and produces code that is shorter and usually easier to read for experienced APL programmers. Because the code is easier to parse and

recognize, especially certain complex design patterns, the interpreters and compilers available for APL can more easily perform certain optimizations that allow for more performance than might seem apparent if the same code were written in a more traditional style in a different programming language (Lochbaum 2018a, 2018b).

Idiomatic APL has arisen from a variety of sources, for a variety of reasons, but the main reasons for writing APL code in an idiomatic fashion are readability (for experienced APL programmers), concision, execution speed, platform independence, data size agnosticism, reliability, and domain scalability.

The original push towards this style comes from Iverson's Turing Award lecture (Iverson 2007), in which he details the following principles for language's designed to foster notational thinking:

- Ease of expressing constructs arising in problems
- Suggestivity
- Ability to subordinate detail
- Economy
- Amenability to formal proofs

These are abstract notions, but as they tend to apply in practice, they manifest as a set of design principles that establish a tension between two ideals and a tendency to prefer the one ideal over the other in most cases of design consideration. These are called the 8 design patterns and anti-patterns of APL (Hsu 2017):

- Brevity over Verbosity
- Macro Perspectives over Micro Perspectives
- Transparency over Abstraction
- Idioms over Libraries
- Data Encoding over Control Flow Embedding
- Structural Information over Nominal Information
- Implicit Semantics over Explicit Semantics
- Syntax Design/Appearance over Semantics

These are not strict rules and do not represent a set of stylistic guidelines that should be enforced, but they do represent a general tendency to be seen within the APL community as to what constitutes "good APL."

Brevity over Verbosity. In many programming language communities, there is a tendency to prefer more verbosity as an avenue towards more explicitness ostensibly to improve the ease with which newcomers to the language may be able to read the code. In APL, there is a strong tendency towards distinguishing code designed for "new users" and code written for general daily use. The one is usually written with a very different, pedagogically focused style, while the other is often explicitly designed without consideration for the new user, in favor of great facility with users of experience (Iverson 2007). This often takes the form of finding shorter, more memorable, and more concise ways of expressing ideas, even if this might come at the cost of hiding certain assumptions (often especially if

it can hide some assumptions that are deemed subordinate) or utilizing the full range and expressive power of a given syntax or primitive. In APL, there is a very real sense in which shorter code is greatly favored over longer code (Whitney 2009).

Macro Perspectives over Micro Perspectives. In APL, there is usually a tendency to want to see the "big picture" of the code and subordinate the syntactic visibility of certain details. This comes in the form of making the global traversals, patterns, or other aspects of the system more syntactically explicit and direct, while hiding some of the lower level details behind implicit behaviors or by making the lower level code shorter (Iverson 2007; Whitney 2009).

Transparency over Abstraction. APL has a very strong tendency towards reduced dependence on library use compared to other languages. While not always seen as a net gain, it is true that the majority of traditional APL code is written with very little emphasis on library abstractions. Much code is expected to be self-contained and complete, without reference to additional dependencies. This has the tendency to elevate the core APL primitives as much more critical in day to day operations, as they are the primary building blocks used by nearly all APL programmers, with only very shallow coverings. This is particularly true the more "sophisticated" the code becomes. Unlike in some languages, the more sophisticated and well known a piece of code becomes, often, the more transparent and less abstracted it becomes, utilizing fewer libraries and exposing more core use of APL.

The result of this is the tendency to prefer direct, transparent representations of solutions to computational problems as opposed to writing additional abstractions over the top of the core APL. Rather than seeing the APL primitives as the core building blocks for new computational concepts, experienced APL programmers tend to use the core primitives as the primary vocabulary for solution exposition, rather than hiding them behind libraries or named functions (Metzger 1981).

Idioms over Libraries. The APL community has long struggled to establish a well-known or de facto library infrastructure, and there is currently no de facto standard source of library code nor any global system for the automatic importation of libraries into APL projects. However, very early on, and continuing forward, the APL community has had a wide and nearly universal practice of sharing code snippets in the forms of idioms. These idioms represent the one-line solutions to certain problems that might arise in various domains. Historically, rather than publishing libraries and documentation for these libraries, the community published books of APL idioms (Pesch 1982). A few the most famous idiom catalogues still exist in digital form (Dyalog 2019b).

When discussing solutions to problems, APL programmers commonly reference idiomatic solutions (that is, transparent solutions that are one line or less long) expressed as simple, direct APL that utilizes no additional libraries or external materials. Such idioms commonly feature a minimum of variables and definitions needed to express the solution.

Data Encoding over Control Flow Embedding. Related to the above, APL programmers tend to prefer code with minimal control flow, where much of the work usually dedicated to control flow is accomplished instead through careful construction of the shapes and types of the APL arrays over which the code computes. Most APL longstanding idioms contain little to no jumping, looping, or even heavy nesting of expressions. Instead, control ideally flows through a single line of APL code without any nesting and minimal name binding or back referencing. A common phrase of Morten Kromberg's, though not original to him, is that "APL is read left to right and executed right to left" (2018). Perlis remarked that this aesthetic possesses similar stylistic principles to natural language (1977). The intent is to have the "most significant, last operation be the first thing you read on the line" (Kromberg 2018).

All this minimization of complex or multi-branching control flow in favor of single directional data flow modeling requires that the logic that would otherwise exist within the control flow of the program lives somewhere else. In APL, it tends to live in the data and shape of the arrays (Bernecky 1999; Eisenberg 1987; Dyalog 2019b; Metzger 1981; Perlis 1977; Pesch 1982; Whitney 2009).

Structural Information over Nominal Information. APL's emphasis on data flow modeling, one-line solutions, and transparency means that naming is generally reserved for highly global and important concepts. This can be seen in the early design of the APL systems, which contained a single top-level namespace together with only flat functions containing a single local namespace, and the continuance of that model long after the same model had fallen out of favor with other programming

language paradigms, with a lexically scoped model of function specification arriving prominently only in the latter 1990's (Dyalog 2019a). Unlike in other systems, APL programmers tend to avoid highly nested structural designs, wherein the various scoping rules have come to play a large role, such as C++, Scheme, and Lisp. In APL, despite having higher-order functions for a very long time, there was never enough consensus or push within the community to deploy first-class procedures at scale, despite many different proposals for adding them into the language.

The idiomatic nature of APL contributes to this tendency, where phrases are more common than abstractive towers. These idioms are globally or relatively scoped and usually avoid any specific naming requirements. They certainly do not require any sort of lexical scoping. In this way, they reveal a bit of their mathematically oriented history (Falkoff 1978; Iverson 2007). In such a space, names are premium real estate and valuable enough not to encourage casual proliferation. Instead, good APL relies more on the structure of the phrase or line of APL to reveal information about the nature of the code, rather than the use of explicit, descriptive names. Some users have advocated for a concept that maximizes the use of variable names as being ideally for the domain or "user" of the software as much as possible, while the structure delivered by the APL code itself reveals the necessary information to the APL programmer (Taylor 2005).

Implicit Semantics over Explicit Semantics. In order to accomplish much of the above, and especially in order to do it in a "whiteboard friendly" way, so that pen and paper could be readily applied to the language as in the original design principles (Iverson 2007), the language and its use

has always favored keeping as many things as possible implicitly expressed in the language, rather than explicitly. The shape of arrays and the reduction operator exemplify this tendency. In other languages, such as statically typed languages, programmers explicitly define and restrict the shape of values, first as ADTs, and then as types to functions, and so forth. Attempts to merge data parallel array-like concepts that mirror APL into statically typed languages have taken this approach as a basic tenet (Slepak 2014; Gibbons 2017). However, APL, as much as possible, encourages syntactically hiding the type and shape of values. A common anecdote about APL expressions tells of the serendipitous application of a given expression outside of the intended use case, where a program was written, for example, to work over some certain type of data, but because the expression was written idiomatically to avoid over-specifying type, shape, or value requirements, it worked on a number of other values and shapes, thus improving the scalability of the expression, which is an aspect of the design principles Iverson coined "suggestivity" (Iverson 2007). Likewise, the design of the reduction operator specifies an implicit, rather than explicit, seed value, called the identity element. In most languages that have something like a reduction function, such as the fold functions in Scheme, there is an explicit seed value. In APL, seed values are implicit, and specific logic in the reduction function ensures the return of an appropriate seed value when applying a reduction over an empty dimension.

This implicit behavior has the overall tendency of improving the brevity of code, but such implicit behavior does not equate to creating abstractive barriers, distinguishing the APL concept of implicit behavior from the library black box concept of abstracted behavior. APL programmers strongly tend towards the subordination of detail but not the literal hiding of it. Implicitness in APL

serves to reduce the use of syntactic space, but programmers still expect the implicit behavior to be well-defined and predictable, in a sense, not visible, but still known. This behavior manifests so strongly in some cases, that some companies utilizing APL in great anger, establish large QA suites designed to catch any case in which the order of floating point operations within any given implicit operation (such as within the runtime of the interpreter) results in an observational change in results at the bit level for their code.[1] Thus, while they take advantage of the implicit behavior to avoid needing to have the core source calculations handle order of operations, they still expect the order of operations to be highly predictable. This emphasis on implicit but predictable behavior is somewhat rare within the programming world.

Syntax Design/Appearance over Semantics. Finally, from early on, the APL cultural emphasized the syntactic clarity and beauty of APL code, rather than its semantic beauty. Many times, Iverson and early implementors chose a more complex semantic model to improve the overall clarity and beauty of the notational expression of solutions, particularly in terms of total economy, rather than a more cohesive semantics at the expense of a more verbose syntax. As a result, typing the APL semantics and syntax in full continues to challenge researchers with the results mostly still unusable for APL programmers on a day to day basis (Slepak 2014; Gibbons 2017), with particular exception to the very APL-like approach taken by Lenore Mullin in defining an axiomatic style semantics for a core subset of APL (1988).

[1] Personal communication with APL companies and Dyalog, Ltd.

"Good" APL expressions emphasize a syntactically memorable, clarifying, and direct solution that highlights the core idea and eliminates as much syntactic "fluff" as possible. This can be seen in the various APL idiom libraries, their emphasis on the single line of code, as well as in the other available libraries, and Roger Hui's famous expression, "monument quality code." Not uncommonly, APL code sometimes lacks the typical semantically constrained rigor of a statically typed functional program, but retains enough correctness, and sufficient syntactic beauty to warrant widespread adoption. Such code may have surprising corner cases or otherwise fail to meet the prevailing standards of "correctness" in some way, but it is valued (sometimes in production) because of its syntactic clarity.

Of course, the very best of APL code possesses syntactic clarity while also defying semantic criticism. As a classic example, code that contains an explicit base case is often thought somewhat "warty" or "ugly." A more desirable solution contains no explicit base-case statement, instead using something of an "inline" base case, but even better is code that utilizes fully implicit behavior to obviate the need for a base case, sometimes at the cost of surprising results on inputs that will not be considered useful. Thus, a classic APL exemplar is an expression that reformulates a problem to eliminate the need for an explicit base case where typically a base case is considered fundamental. This is often done as one of the requirements to refining certain APL expressions (Scholes 2014).

Fundamental Stylistic Rules. The above design principles and patterns helps to create a context for understanding idiomatic APL code, but what of the specific stylistic artifacts that emerge?

Unfortunately, no strict style guide exists that indicates a universal agreement on what constitutes "idiomatic" APL because of the subjectivity of human judgment and style. However, there are enough overlaps in common features that appear across a range of common styles to highlight some common patterns.

Among the first is a strong tendency towards "spiritually functional" code. This is code that, at the low-level and at its heart, expresses most of the computational requirements in terms of functional expressions without "micro side-effects." That is, effects that appear at the leaves of the source tree. Instead, good APL tends to use large, bulk side-effect operations at the top of the computational tree, rather than within the "inner loops" (Dyalog 2019b; Pesch 1982; Eisenberg 1987; Metzger 1981).

The concept of semantic density also appears frequently. Good APL often prefers names that possess semantic meaning within the specification domain of the problem, rather than names referring to abstractive intermediate steps between the core APL vocabulary and the problem domain. This work follows this practice by expressing various tree transformations as native APL expressions without many additional variables or namespaces. Those variables that are used are as globally relevant as possible and generally refer to explicit concepts at the level of the tree. This contrasts with other treatments on the same subject that abstract over the underlying implementation language and create at least one intermediate layer between the core programming language and the domain of tree manipulation (Taylor 2005; Ren et al. 2012; Goldfarb, Ju, and Kulkarni 2013). In APL, good style generally correlates with preventing intermediate levels of abstraction.

As a result of semantic density considerations (Taylor 2005), programmers will sometimes avoid excessive variable names if they can. That is not to say that they are not used, but the more the code can be made readable without additional names, the better. If a programmer can replace a variable name or function binding with a short, idiomatic expression of the same concept, many would prefer the idiomatic expression except in the most pedagogic circumstances or in companies with large turn overs that require almost constant pedagogical treatment.

Likewise, loops, iteration, and other sorts of logical "jumps" in the control flow are usually avoided as much as possible. A single iteration on a single line that serves as an important semantic point at the level of the domain of the problem is okay but is generally to be avoided. Any looping within a treatment that doesn't have a direct correlation to the expression of the problem in domain terms is generally avoided or to be rewritten to avoid explicit looping (Scholes 2014).

Good APL tends to avoid the "individual element" as much as possible (Kromberg 2018). That is, the emphasis is on using the APL primitives (which are almost all defined over bulk elements and bulk operations) to operate on either whole arrays at a time, or on large sub-sections or regions of a given array at a time, not on individual elements at a time. Thus, if one can avoid indexing, it is usually best to do so, but if indexing must be used, it is usually best to index multiple elements at a time, as many as makes sense.

Likewise, avoid indirection: instead, find the smallest, most direct solution to the problem that avoids indirections such as function abstraction (Iverson 2007). In APL, the term "code smell" often applies to code that uses too many "dfns" or functions, as well as code that is heavily nested, relies on

lexical scope very much, or that has any other structural elements that increases the depth of the tree outside of straight-line data flow expressions, and even in those, the goal is to avoid too many parentheses. This usually means that good variables should have very short lifespans or be global in nature, and that the data should ideally flow through a program statement from right to left without stop.

This text makes some concessions towards readability for those not fluent in APL; some source code uses overly verbose and unnecessarily tedious syntax. However, the structural repetition in these cases has been deemed worth the extra cost in order to highlight a specific point and thus its use. While some minor and theoretical performance penalties arise because of this, potentially inhibiting certain parallelization efforts on low-analysis platforms such as interpreters, the nature of the code is such that these benefits are not worth the pedagogical cost, though they may cause issues in commercial application of these algorithms at scale.

In general, however, the code here has as classically canonical a style as the author is capable of, in order to demonstrate and highlight the use of such code. This “idiomatic style” is an important aspect to the design of the compiler itself, because this style significantly enables the cross-platform performance and relatively low analysis burden necessary to achieve high performance across multiple platforms. That is, idiomatic APL style benefits not just human consumption but also contributes to the high-performance results in traditional APL code on interpreters and other historically challenging systems. The following sub-section will explore this idea in more detail.

2.1.5. Performance

Even in Iverson's original work (2007), data parallelism represented a significant design element of APL. Early vector machines sometimes leveraged APL as the high-level language of choice due to the natural fit between APL and vector architectures (Schwarz 1991; Budd 1984; Ching and Katz 1994). It is the design and practical use of the APL primitives that primarily enables this synergy, since, as previously noted, APL primitives are almost entirely data parallel. If one considers certain primitives that provide meta-information as still "array at a time" operations, such as the depth or shape functions, then the entirety of APL as a primitive vocabulary is data parallel friendly. Since the primitives operate over large bulk regions of array data, they readily admit implementations on fine-grained SIMD parallel hardware.

All APL scalar primitives are defined in a SIMD fashion. The translation of such primitives into data parallel counterparts is trivial, at worst. They have constant time critical paths. However, the parallel implementations of the mixed primitives do not so easily reveal themselves, and they deserve further examination. Consider the summary analysis in Table 1 and Table 2, which highlights the critical path complexity of each APL primitive. The parallel implementations of these primitives exist widely in existing literature, so much so that highlighting any specific implementation as canonical is somewhat difficult, because the primitives often represent areas of research around a specific computation. Thus, the tables above highlight their computational research class, rather than a specific implementation, since several implementations exist.

Mixed Function	Critical Path	Computational Class
Reshape	Constant	Data Movement
Ravel	Constant	Shape Change
Catenate	Constant	Data Movement
Table	Constant	Shape Change
Reverse	Constant	Data Movement
Transpose	Constant	Data Movement
Mix	Constant	Data Movement
Split	Constant	Data Movement
Enclose	Constant	Shape Change
Partitioned Enclose/Partition	Logarithmic	Data Movement
Enlist	Constant	Data Movement
Pick	Constant	Data Movement
Take	Constant	Data Movement
Drop	Constant	Data Movement
Replicate	Logarithmic	Scan
Expand	Logarithmic	Scan
Without	Logarithmic	Set Operations
Intersection	Logarithmic	Set Operations
Unique	Logarithmic	Set Operations
Union	Logarithmic	Set Operations
Same/Left/Right	Constant	Identity
Index Generator	Constant	Data Generation
Index Of	Logarithmic	Search
Where	Logarithmic	Scan
Interval Index	Logarithmic	Search
Membership	Logarithmic	Search
Grade Up/Grade Down	Logarithmic	Sorting
Deal	Logarithmic	Random Number Generation
Find	Logarithmic	Search
Shape	Constant	Shape Change
Depth	Constant	Data Movement
Match	Constant	Data Movement
Tally	Constant	Shape Information
Decode/Encode	Constant	By Definition
Matrix Inverse/Divide	Logarithmic	Linear Algebra
Indexing	Constant	Data Movement

Table 1. Mixed Primitive Functions and Their Critical Paths

In addition to general research into parallel implementations of the above computational classes, APL vendors have analyzed and discussed the performance of APL code and their optimized implementations of key primitives and idioms. One note of worthy discovery was the relatively small data sizes used on many variables in traditional production APL. This discovery is one reason that APL on the GPU has not received the attention from APL vendors that it might have in the past. In traditional APL code, without structured programming statements or dfns syntax (both invented in the 1980's and 1990's), much of the APL code in existence contained too much iteration and looping to be able to achieve high parallel performance. In code written in this style, the analysis requirements prohibit good parallelism, as is the case in other programming languages. As a counterpoint, APL code written in a more idiomatic style with better use of bulk operations and larger array data sizes enables a much greater degree of parallelism at much less cost of analysis (Hsu 2014, 2015).

A unique feature of APL's performance results has been the high performance that APL has been able to achieve relative to traditionally high-performance programs. This result has been noted in a variety of fields, including finance (Whitney 2009; Lochbaum 2018a). Ironically, these gains

Operator	**Critical Path**	**Computational Class**
At	Constant	Data Movement
Commute/Compose	0	Pre-execution
Each	Constant	Data Parallel Map
Inner Product	Logarithmic	Matrix Multiplication
Key	Logarithmic	Sorting
Outer Product	Constant	Data Parallel Map
Rank	Constant	Data Parallel Map
Reduce/Scan	Logarithmic	Tree Reduction/Parallel Scan
Stencil	Constant	Data Parallel Map

Table 2. Primitive Operators and their Critical Paths

appeared using fully interpreted implementations compared to fully optimized and compiled code written in languages such as C. In these benchmarks, both at a micro level and at the application level, the results have consistently demonstrated interpreted APL's very high speeds on applications requiring such throughput. In some cases, the performance achieved is truly astounding, even discounting the interpreted code base (Lochbaum 2018b). Idiomatic APL code utilizing high-performance primitive implementations enables such code to incur minimal interpretive overhead relative to the amount of time spent "crunching" within the interpreter runtime, greatly improving the overall performance compared to typical expectations (Whitney 2009).

Other areas of related work that demonstrate the performance of such techniques is the application of linear algebra to graph processing (Kepner and Gilbert 2011) and the use of vectors within column store, in-memory databases, many of which use APL or array-based languages (Whitney 2009; Hallenberg 2016). In these cases, as in previous ones, the use of idiomatic APL style helps to eliminate complexities of code analysis and execution overheads, while maximizing the time spent in highly tuned runtime libraries.

Idiomatic style only goes so far with an interpreter though, and modern data sizes and memory behaviors have resulted in significant bottlenecks that can appear on truly high-memory bandwidth applications. The research into fusion strategies has greatly improved the quality of code that can be generated by APL compilers. APL specific research has demonstrated that idiomatic APL code already contains enough type-level information implicitly to detect many of the classical type errors around rank and shape that are the purview of various type systems written around arrays (Bernecky 1997).

Additionally, traditional APL compilers have demonstrated the relative ease of compiling APL expressions into fused operations (Bernecky 1997, 1999; Bernecky and Scholz 2015; Budd 1984, 2012; Ju, Ching, and Wu 1991) that prevent certain memory bottlenecks that are the bane of interpreter performance. This includes the remapping of memory allocations in order to facilitate better internal loop generation of the compiled code. In addition, there are now runtime solutions to help assist with the fusion problem (Malcom 2012), these runtimes make it possible to implement APL primitives and leverage fusion both at runtime and at compile time. They rely, however, on the expression of the computations in a shape and form that mimics idiomatic APL code.

Thus, without the use of idiomatic style, more traditional code styles require much more analysis and prove difficult to execute efficiently in parallel. However, idiomatic APL code significantly reduces the analysis burden so that the majority of the performance bottlenecks are removed outright, and those that remain can be readily eliminated at compile time (Budd 1984; Bernecky 1997; Hsu 2014, 2015) or runtime (Malcom 2012; Schwarz 1991) to enable high-performance, data parallel execution of APL code.

The use of idiomatic APL code also affects the ease of asymptotic performance analysis, but this is best discussed after a thorough discussion of the compiler construction itself. Section 5.3 addresses these considerations in more detail.

A final aspect of the synergy between idiomatic APL and performance comes in the data model. When implementing APL, most systems (the author has been unable to source a single exception to this) store the value data as a single large, contiguous region of memory in row-major

order. Some systems will store the shape data contiguous to this and others store it separately. However, for the purposes of data parallel performance, this contiguous allocation of memory is critical. When executing idiomatic APL, most computation operates over whole arrays at a time, which means that the access patterns of most operations follow a perfectly sequential access pattern, the pattern that is fastest on most modern CPUs and GPUs.

In cases where this is not the case, even bulk indexing provides an advantage on CPUs and GPUs where that memory appears in strided format, and on GPUs, the random access indexing is still done in bulk, which can be done more quickly on GPUs than CPUs, and can potentially be done in an ordered form that improves performance. This prevalence of strongly contiguous reads and writes within idiomatic APL contributes to real world throughput. This enables the use of SIMD vector operations with minimal overheads due to random access memory patterns that are otherwise very common on GC-heavy, tree-oriented ADT object models. In the case of tree manipulations, which are often memory, not CPU bound, this plays an even greater part than more computationally intense problems.

Idiomatic APL's use of contiguous, large memory regions also significantly reduces garbage collection overheads. In fact, garbage collection is not necessary for such a model and a stack-based liveness model suffices. Thus, idiomatic APL incurs zero garbage collection overhead. In allocation heavy operations, such as functionally defined tree transformations, this represents a significant source of performance improvement, as discussed in future sections.

2.2. The Co-dfns Language

The late John Scholes introduced the dfns syntax for user-defined function specification in the 1990's, with incremental refinements to the semantics and implementation correctness throughout the following decades (Dyalog 2019a). It extends the basic APL expression syntax with a function specification syntax that allows users to write anonymous functions with error and conditional guards. These functions may be monadic or dyadic first order functions or second order.

The syntax is extremely simple. A function is simply a matched pair of curly braces surrounding zero or more statements to be executed in order from left to right, top to bottom. Each statement is either an `Expr`, `FnBind`, Guard, or Error Guard. For the definition of an `Expr` or `FnBind`, see the previous section 2.1. Each statement is separated by either a newline or a statement separator `⋄`. A guard and error guard both consist of two expressions separated by, respectively, a colon `:` or double colon `::`. A guard is a one armed if statement where the first expression on the left of the colon should evaluate to a Boolean scalar value. When executing a guard statement, the left expression is executed first. If the value is 1, then the dfns returns the result of executing the expression on the right of the colon. If the value is 0, then the expression to the right of the colon is not executed, and execution proceeds to the next statement in the dfns. An error guard works the same way, but it serves as a barrier for errors in execution. The expression on the left evaluates to a vector of integer values indicating error codes to be trapped, where a zero represents any error. As execution proceeds through the dfns after the error guard occurs, if one of those expressions signals an error code matching the error guard and no error guard after it, then the dfns will return the result of executing the error

guard's right expression. When executing statements, if a statement has as its last operation a binding operation, then execution proceeds to the next statement in the dfns, otherwise, the result of executing that statement is returned as the value of applying the dfns.

Each dfns implicitly binds the values ⍺ and ⍵ to its left and right arguments. If the operands ⍺⍺ or ⍵⍵ are used as free references within the body of a dfns, then the dfns is an operator, and the left and right operands are bound to ⍺⍺ and ⍵⍵, respectively. Additionally, the function ∇ is bound to the dfns' function value and ∇∇ is bound to the dfns operator value if it is an operator. In the cases where one desires to apply a dfns with an optional left argument (ambivalently), then the variable ⍺ may be bound in the body of the dfns. In this case, if ⍺ already has a binding at application time, then that binding will be used, but if not then the binding given in the body will be executed when it is reached during execution of the dfns. Here are some example dfns:

```
:Namespace

factorial←{     ⍝ Tail recursive factorial.
    ⍺←1
    ⍵=0:⍺
    (⍺×⍵)∇ ⍵-1
}

fibonacci←{     ⍝ Tail-recursive Fibonacci.
    ⍺←0 1
    ⍵=0:⊖⍴⍺
    (1↓⍺,+/⍺)∇ ⍵-1
}

⍝ Quick Sort
⍝ https://www.dyalog.com/blog/2014/12/quicksort-in-apl/
Q←{1≥≠⍵:⍵ ⋄ S←{⍺≠⍨⍺ ⍺⍺ ⍵} ⋄ ⍵((∇<S),=S,(∇>S))⍵[]⍨?≠⍵}

:EndNamespace
```

The only other addition in the source input to the compiler is an explicit top-level namespace utilizing the Dyalog “Namespace” syntax, as the above example demonstrates.

The dfns syntax itself is lexically scoped and functions may be nested. Free variables in the body of the dfns resolve according to lexical scoping rules. None of the compiler code that requires a thorough understanding of these rules, as it does not leverage lexical scoping to any significant degree. However, the compiler does need to address lexical scoping appropriately for input programs, and the section on lexical resolution in Section 3 details more information on the semantics of lexical scoping.

2.3. Nanopass Framework

The Nanopass compiler framework (Keep 2013) is a syntactic and library framework for designing compilers around a specific model of small compiler passes strung together in a data flow manner, each making small transformations to the structure of a tree to eventually complete the compilation process. Even though the compiler described here bears no syntactic resemblance to the Nanopass compiler design, the framework itself has significantly influenced the design of the compiler and it serves as a benchmark to compare against the compiler.

In particular, the compiler itself relies on an architecture of small conceptual operations that work one after the other on the AST until the result is suitable for assembly. The small transformations exhibit data flow design in a similar style to Nanopass compilers, but without the inherent procedural boundaries used by Nanopass compilers.

However, the Nanopass compiler takes a significantly different general approach to specifying and expressing compiler design compared to the compiler described in this work. Specifically, the

framework itself provides for two major syntactic abstractions over the underlying Scheme/Racket language. It provides for a convenient syntax for specifying intermediate representations in the compiler. This is a convenient wrapper around the record type system of the underlying implementation and provides the additional benefit of allowing for intermediate representations to be specified as the difference between the new representation and an already defined representation.

The core of the syntax is a pattern matching abstraction that defines a transformation over the structure of one intermediate representation (called a language) to another intermediate representation. This transformation is defined through various pattern matching clauses, and the syntax provides significant additional assistance, including the ability to automatically dispatch trivial pattern matching cases (such as identity transformations) as well as enabling automatic recursive decent (depth first traversal) through the use of cata-morphism forms and other such syntactic niceties.

Added to the above syntactic benefits, the implementation provides for relatively sophisticated data integrity checks that ensure the transformations are reasonably well typed, and it does much of this analysis at compile time. This gives the compiler writer some of the benefits of a statically typed programming language from within an otherwise "untyped" language such as Scheme or Racket.

Andy Keep demonstrated that the performance of the Nanopass framework was sufficiently good that it did not seriously impact the performance of the Chez Scheme compiler (Keep 2013), which was known for its very fast compile times. This was an important result in the exploration of compiler design, as it demonstrated both that the use of many small passes in the compiler as well as the use of

additional syntactic support did not necessitate a proportional degradation in the performance of the compiler design over more traditional mono-pass or heavyweight pass designs that had less syntactic abstraction.

The benefits of the Nanopass framework over other, more traditional approaches is the relative ease of working with the passes and the overall design of the compiler. Because the passes are so small, they are easier to understand and debug. Because the passes are decoupled from one another, it is easier to design the compiler around how the passes themselves interact, rather than having their effects intermingled in larger passes. This allowed for some of the benefits of functional programming to be applied to the design of the compiler architecture. This had the effect of making the compiler simpler and easier to check, without significantly increasing the cost of execution.

3. COMPILER

3.1. Overview

The Co-dfns compiler follows the same spirit as a Nanopass compiler design. That is, it emphasizes a single linear data flow where the input AST undergoes a series of transformations from IR to IR, each IR differing slightly from the previous one. Each pass is meant to be small and easy to digest. From there, the underlying implementation techniques used to implement each compiler pass and the implementation of the Nanopass design eschews most of the features utilized in the formal Nanopass framework. In particular, the Co-dfns compiler consists of a single function composed of a series of APL expressions logically grouped into individual compiler passes. The design dedicates no code to the definition of ADTs or record-type abstractions, and it uses no abstractions on top of the core APL language. Instead, an inverted table directly encodes the IR, and each compiler pass idiomatically transforms the inverted table's data. Without creating an abstractive layer to represent tree operations on top of core APL, the Co-dfns compiler directly maps raw APL idioms into the tree transformation domain. These two design decisions contrast sharply with the explicit DSL model used by the Nanopass framework that introduces explicit `define-language` and `define-pass` forms on top of the core Racket/Scheme language for the specification of intermediate ASTs and compiler transformations.

Architecturally, the Co-dfns compiler consists of the following passes applied in order over an AST:

1. Convert the AST to utilize a parent vector representation
2. Compute the nearest lexical contour for each node
3. Lift functions
4. Wrap expressions
5. Lift guard test expressions
6. Count the rank of indexing operations
7. Lift/flatten expressions
8. Assign a specific frame slot to each variable in each function body
9. Arrange the function body frames according to their depth in the lexical stack
10. Record the exported names for each module/namespace
11. Resolve variable references lexically to a specific frame and slot

The following section will explore each compiler pass in detail, but it is important first to understand generally the AST involved as well as the way in which the Co-dfns compiler encodes and works with the AST in source.

Each compiler pass in the Co-dfns compiler demonstrates a set of general tree transformations, and together they form a template for building other transformations. This section explores these operations in detail, but the summary of these operations is as follows:

Computing Parent Vector	Shows the relationship between depth and parent and the use of ⌸ to operate over different levels of the tree.
Compute Nearest Lexical Contour	Basic traversal pattern; how to use inverted tables for efficient record extension and lookup efficiency

Function Lifting	Basic manipulation requiring nodes to be relocated to other parts of the tree; efficiently adding new nodes to the tree; managing tree hierarchy without the use of a call stack/recursion
Wrap Expressions	Node splicing; inserting nodes as new parents into the middle of a tree; more advanced selection of nodes with specific properties, that is, "sub-tree" selection
Lifting Test Expressions	Splicing nodes as new siblings; managing record field deletion; leveraging ordering of an inverted table; node exchange
Count Rank of Indexing	Working with children as a group; working with collections of nodes using ⍇ over parent-child relationships
Expression Flattening	Managing complex node permutations; manipulating sub-trees independently; connecting order and tree edges
Computing Slots	Utilizing more complicated auxiliary structures; analysis over parts of the tree; working with variables and symbols
Computing Frames	Working with multiple edge structures over the same nodes; working with depth via pointer vectors
Computing Exports	Preserving data that might be lost through externalization; basic tree selection, querying, traversal, and accumulation
Lexical Resolution	Complex traversal patterns; environments; lookup; managing memory bandwidth, parallelism, and apparent serial blockages; working with different views of the AST

Together, these passes demonstrate all the necessary elements of working with and manipulating trees to perform arbitrary analysis and modification of the structure of the tree.

3.2. The AST and Its Representation

At the heart of any compiler is its IR or AST representation. Many modern compilers utilize small IRs that differ only slightly from one another to simplify pass construction. For the most part, compilers and other tree operations in a similar class strongly favor the use of ADTs or some other recursively defined record structure. The underlying language usually dictates the specific form that such structures take, but the canonical approaches use the type system in typed functional programming languages, class hierarchies mirroring the same in OOP languages, and either structs or record types in C-like or untyped languages such as Scheme or Python. While the supporting syntax for creating these structures differs from language to language, the underlying in-memory representation remains markedly similar across implementations.

This record-type representation of trees generally shares a specific set of common traits no matter what language is used to create them. They generally allocate on the heap, use atomic or unit level records or objects to represent nodes in the AST that contain the appropriate fields for that node, and use pointers or reference semantics to represent edges between nodes. Usually the system will allocate/destroy each node independently, modulo some adjustment of any shared linkages to maintain the integrity of the tree. Furthermore, a single root node serves as the point of entry and primary reference for the tree when passed as an argument to functions. While an algorithm may at times hold a reference to a sub-tree of the whole tree, this is usually a short lived operation, and most compiler passes ensure that they accept a whole AST as the input and return a whole AST as the output, referred to by its root node. Common in such algorithms link the parents to the children as

opposed to children linking to their parents. Some scientific or graph applications prefer to link children to the parents, but in transformation algorithms that rearrange the tree each non-leaf node usually has some list of its child nodes, rather than a child having a pointer back to its parent node.

In order to compare this representation to other possible representations, the desirable qualities of any given representation should be enumerated. For the purposes of a compiler or tree manipulation heavy program, the following qualities are all desirable:

1. Easy to traverse
2. Easy to manipulate: add, delete, update
3. Easy to alter the definition
4. Easy to type-check
5. Efficient in memory and computation time

3.2.1. Record-type Representation

The traditional record-type representation exhibits several favorable qualities for popular languages and programming paradigms that make certain assumptions about the execution model of the machine.

Ease of Traversal. The record-type presentation of an AST fits perfectly onto the well-studied and well-used paradigm of structural recursion. In languages with good support for heap-allocated objects and reference types, this representation allows for a recursive description of a traversal that naturally mirrors the top-down bias exhibited by the structures usually used in compilers. Furthermore,

sophisticated pattern matching libraries and syntaxes specifically target these traversal patterns. They ease the matching and extracting of information from these record-type representations and simplify many of the most common traversal patterns albeit at the cost of a special syntax for handling such objects.

The use of ADTs and record-type representations so pervades functional programming and OOP that it represents the predominant method for working with most domains. In such languages, representing trees in the same way allows the use of the many syntactic and mechanical aids provided by languages to work with such structures.

Ease of Manipulation. Adding, deleting, and updating nodes in a record-type representation is usually quite simple. In garbage collected languages, constructors permit trivial allocation of new nodes, while libraries for other languages provide ready-made abstractions for the same. Likewise, in a garbage-collected language, deleting a record is trivial as well, since the code simply removes any references to the node from the tree and lets the garbage collector handle the rest.

When updating the tree, some methods that emphasize a monolithic approach utilize mutation over the structure to change the tree, while the functional approach, commonly favored even in imperative and OOP languages, constructs a new tree derived from the old tree with the appropriate changes made to the new tree, after which the code deletes the old tree. While this incurs some memory overheads, often, the language runtimes optimize the allocation of many small objects, greatly reducing any cost for allocating a new tree. Together with the garbage-collector, this generally

allows the easier to program functional approach to compete with the mutation-based approach, so much so, that programmers using otherwise imperative or OOP language often greatly favor the functional approach.

Thus, while update is not as easy in some languages for the record-type representation as it might be, real world use of the record-type representation often ignores update in favor of whole tree copy.

Ease of Typing and Definition. Many languages provide built-in support in the type system and the core language for building and defining ADTs or recursively defined objects. This can make it easy to create different styles of ASTs, or IRs, such as provided by the Nanopass framework, which enables the programmer to derive a new IR from a previously defined IR. The programmer then benefits from improved type checking and error handling when constructing ASTs and should the programmer attempt to create a structurally incorrect AST, the system provides a suitable, hopefully easy to read, error either at run time or compile time.

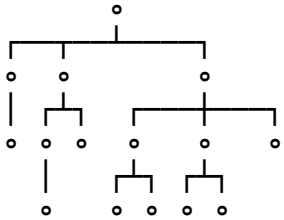
Languages such as Haskell provide for significant amounts of static guarantees around ADTs through heavy use of type-level features.

Efficient memory and computation. The record-type representation relies primarily on pointers as its core building block, since all nodes are constructed and linked together primarily by pointers to more primitive types or to other node types. This is not the most efficient use of memory, and so there

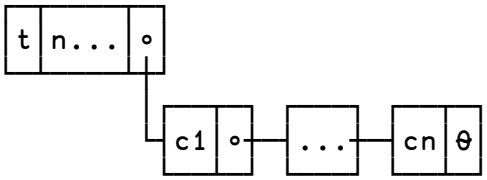
is usually a bit of excessive overhead in terms of memory consumption per edge and per node in a tree, but traditional wisdom considers this overhead relatively minor. Regarding computation, compiler passes operating over a record-type representation primarily chase pointers. Traversing the AST requires following a chain of pointers to disparate and usually segmented or fragmented regions of memory. The costs associated with these passes center around traversing the tree, creating new nodes and edges to form the next tree, and in comparing values to find the right parts of the tree on which to work. CPUs with branch prediction, pre-fetching, and other logic try to optimize this sort of operation and programmers often consider the resulting performance good enough. This is particularly true of systems with good garbage collection and heap allocation runtimes. In fact, these runtimes provide such a performance improvement over other approaches that runtime and language implementations that take advantage of this for tree processing sometimes consider these optimization features, not just programmer conveniences.

Notwithstanding the excellent and desirable properties that the record-type representation exhibits when examined through the lens of traditional programming languages on a CPU architecture, the execution of tree manipulation algorithms on a vector-oriented machine (including some modern CPUs that focus heavily on vector parallelism) presents serious limitations along these five desirables due to the change in underlying architecture.

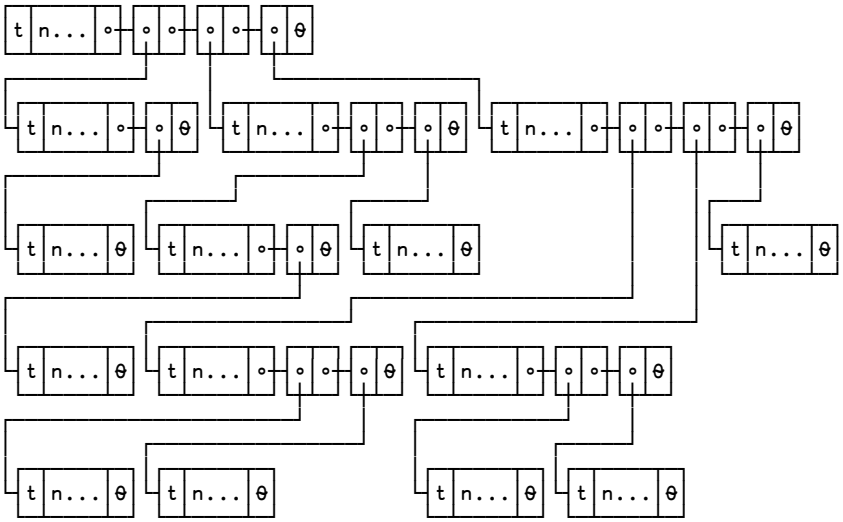
In order to better examine these issues, consider the following small tree:



Note a few of the important features here often found in the ASTs used by compilers. The tree is unbalanced, of variable branching factor and depth, and for a given node, one may or may not know the number of children. To represent this tree in a record-type system, it would be common to have a type/sub-type header, some additional metadata for each node, and then a list of the children of the node. This list of children is very often stored as a linked list, though a vector is sometimes used as well. This gives the following structure for an individual record:



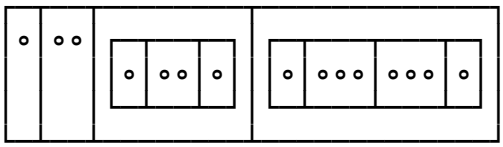
The above record-type representation yields the following in-memory representation of the above tree:



The above depiction illustrates starkly some of the challenges in using the record-type representation on vector machines. Note that each block of allocated memory is not guaranteed to be contiguous with any other block, and that most values are pointer values, which requires the use of whatever size pointer is native to the machine. On most current machines, this is 64-bits for each pointer value. Observe the need to construct a pointer pair for each edge in the tree, and, assuming aligned memory accesses and only a single 64-bit value as the data for each node, that each node requires 3 64-bit values, or 192 bits. This leads to a total memory requirement for a given tree as `(N×192)+E×128` bits, where `E` and `N` are the number of edges and nodes, respectively.

Using arrays to store the children instead of linked lists reduces the cost to 128 bits per node and 64 bits per edge for children. Inlining the child list as the tail of the node record using arrays, then the representation further compresses to require only 64-bits per edge plus the cost of the node including the count of the number of children. Using this compressed array representation, the APL expression for the above tree looks something like this without the `n` fields:

```
      t←'∘'
      ⊢ast←t(t t)(t(t t)t)(t(t t t)(t t t)t)
```



Note that this array-based compression loses some of the original flexibility of the record-type representation, since it requires more effort to change the number of children for a given node.

On architectures that amortize the costs of random access to memory, have high clock speeds, and can handle branching code paths effectively, the record-type costs are not particularly high,

especially considering the additional benefits gained from being able to use a garbage collected language with sophisticated support for ADT constructions.

The requirements differ significantly for vector machines. All GPUs, and many CPU architectures, require effective utilization of the vector instructions and the provided SIMD programming models in order to achieve peak performance. The SIMD or vector computation model is characterized by a single operation across multiple lanes or threads of computation, known as single-instruction, multiple data. Since the threads are designed to operate in lockstep across multiple pieces of data, branching code that results in divergent code execution paths for different data points usually results in severe performance penalties or not being able to apply vectorization at all. Furthermore, many vector machines do not dedicate significant hardware to pre-fetching, pipelining, or the like, thus penalizing random-access memory reads and writes. Even on CPUs, despite the pre-fetching algorithms, a severe memory penalty exists for accessing memory in a random pattern rather than accessing memory using a sequential pattern. Additionally, many vector operations work best over small element sizes, meaning that the number and size of contiguously allocated elements limits the amount of available parallelism. For instance, many of the vector instructions on Intel CPUs rely on fitting multiple small values into a single 128-bit or 256-bit memory region. On GPUs, it is exceptionally difficult to encode recursive, branching algorithms in a manner that fully leverages the hardware capacity. On CPUs, while they adequately support recursive, branching computations, the presence of recursion and branching interferes with efficient vectorization of such algorithms.

Efficient garbage collection, particularly fast allocation and deallocation with fragmented memory spaces, remains an open and difficult problem for GPUs and other vector-heavy architectures. This makes the use of garbage collected languages on GPUs essentially intractable at the current time.

This leads to the following list of desirable features for vector programs:

1. No recursion
2. No branching
3. Small integer/floating point values
4. Contiguous data allocation
5. Sequential read patterns over memory
6. Minimal memory requirements
7. Minimal data-dependencies in program control flow
8. Use of SIMD/Data-parallel operations

The record-type representation seems much less desirable when examined in this light. The features that enabled ease of manipulation, traversal, type checking, and definition are all things that make it unsuitable for use on vector machines, namely, garbage collection, recursive control flow patterns with pattern matching, and reference semantics utilizing heap-allocated pointer heavy objects.

In terms of time and space usage, the record-type representation also begins to struggle when applied to vector machines. The need for space savings increases on GPUs. To execute efficiently on vector machines, the need for small data types to maximize parallelism coupled with memory access and alignment restrictions means that the relatively heavy use of pointers in the record-type

representation precludes the effective use of vector instructions. The non-contiguous memory access patterns in tree traversals hampers effective vector-parallel execution without significant restructuring and rewriting, a difficult and challenging problem.

3.2.2. Linearizing Relational Representations as Tables

What other representations might solve this problem? Alternative representations must first address contiguous memory allocation. Linearizing or serializing a tree into a format with a specific, concrete in-memory representation that preserves data locality removes the issue entirely.

The record-type representation above is a simplified method for representing relational data, and the term "record" signals the related nature of relationally stored data and a tree represented as records. A record-type representation can be modeled as a set of tables with foreign-key references to various nodes in each table with each node given a globally unique reference id by which to refer to the node.

The relational model does not specify a specific on-disk representation, but a matrix of 3 columns trivially represents such a relational model. The first column encodes the edges in the node somehow, the second column indicates the type, and the third column contains the field data for each record. In this case, because a matrix is an ordered structure, the row associated with each node can be used as the reference id for that node.

The question then becomes one of choosing the right representation for the first column of the matrix, which indicates how each node is connected to another in the tree. This column, which might be a vector or a higher-ranked array, represents a linearization or serialization of the node-edge

information stored in the tree. It provides a specific order to the edge information that was otherwise stored in disparate regions of memory in the record-type representation.

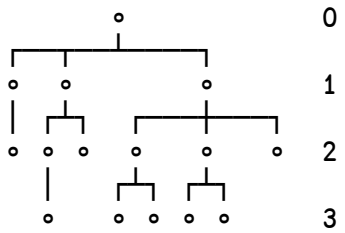
Linearization of trees and their representation has a long history, with Kenneth Iverson exploring such topics in this 1960's era APL book (1962). There he explored several different representations for linearizing a tree, but through the years two common linearizations have appeared.

One, treated here briefly, stores the type code of each node in the vector and orders the nodes in the matrix in their depth-first, pre-order traversal. The root node is the first node and the left-most child of the root node is the second element in the matrix and so forth down the tree. If each node has a static record field count, this sufficiently encodes all the information about the tree's structure. Traversing the AST by reading along the vector and recursively applying the appropriate constructor for each type reconstructs a record-type representation of the tree. This representation requires that all the records have static size including the listing of children.

3.2.3. Depth Vector Representation

Another possibility is to order the nodes in depth-first, pre-order traversal as above, but to use the depth of the node instead of the type of the node in the vector. This depth vector encodes the information about the edges in the tree, but also allows for variable length record types.

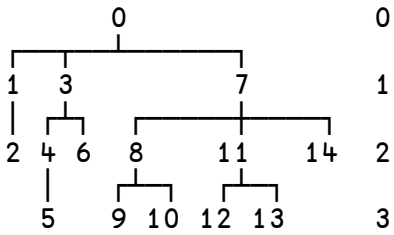
As an example, recall the previous sample tree, with the depths listed on the right:



To represent this tree as a depth vector requires no knowledge of the node types, as follows. This representation stores the type of each node separately:

```
d←0 1 2 1 2 3 2 1 2 3 3 2 3 3 2
```

The structure of depth vectors depends on ordering the elements according to a specific traversal pattern, in this case, depth-first pre-order traversal. The id of a given node in the tree is the position of that node in the associated depth vector or other linearization. Thus, the ids of each of the nodes in the above tree are given in the following; note the traversal pattern:



Observe, then, the node ids with the depths of each node with depth first ordering:

```
      ⍉(⍳≠d),⍪d
0  1  2  3  4  5  6  7  8  9  10  11  12  13  14
0  1  2  1  2  3  2  1  2  3   3   2   3   3   2
```

Within the Dyalog APL community this depth vector represents a canonical encoding used to represent tree data such as that found in JSON objects and XML data. It has several favorable properties over the record-type representation when compared for vector machine friendliness.

A depth vector requires very few bits per node to represent the AST, since it encodes the depth of the node, rather than the id of the node. Thus, each element requires only the number of bits required to encode the maximum depth of the tree. Since the depth of a tree tends to be much smaller than the number of nodes in a tree, this tends to be significantly smaller than the 128 bits per edge plus 64 bits per node used by the record-type representation.

Furthermore, a depth vector ensures contiguous memory access and groups sub-trees contiguously as well, so that functions over sub-trees also operate with a contiguous memory region. If a traversal over the AST matches the depth-first pattern, which it often does in compiler traversals, good vectorized, non-recursive, non-branching solutions to many traversals exist. This makes certain read patterns very fast over depth vectors because of the low memory overheads and the fast vectorized algorithms.

However, not all is well with depth vectors. Two major issues arise when working with depth vectors for compiler design. First, constructing a new AST or modifying an old to add new nodes may require significant copying of the tree in order to make space in the vector for a new node in the middle of the vector. Moreover, many algorithms are difficult to express without the use of nested vectors, which are another form of tree data that is difficult to traverse efficiently on the GPU. Finally, the depth vector efficiently encodes the edges of the tree using an explicit ordering between elements in the vector, which introduces a data dependency that may require searching the entire vector in order to find a node's parent or a node's children.

In short, while the depth vector representation exhibits favorable space usage and traversal ease, it does not permit easy manipulation or modification because of excessive copying or traversal requirements to enforce or derive parent-child information. Recursive or iterative algorithms may address these concerns on CPUs, but this hampers vectorization and introduces the same issues as the record-type representation when used on vector machines.

Nonetheless, depth vectors are simple to understand, very space efficient, and relatively easy to construct using recursion. The Co-dfns parser uses parsing combinators to construct the AST from text to a richer structure and so the depth vector works well in the parser. The parser produces the AST using a depth vector to represent the tree structure, and this AST is passed to the compiler. The following expression converts a record-type representation to a depth vector representation:

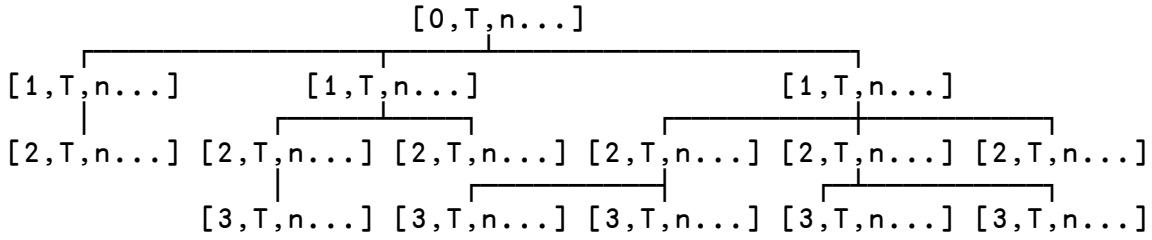
```
      ∊0{(⊢,(⍺+1)∇¬)/⌽⍺,1↓⍵}ast
0 1 2 1 2 3 2 1 2 3 3 2 3 3 2
```

Note that the above recursive expression does not utilize a base case and suggests a natural parallel execution use a tree-reduction strategy. The following sections explore optimizations and representations that reduce the vectorization issues with depth vectors.

Linearizing the tree structure has an additional benefit. The record-type representation groups its memory allocations and layouts based around individual nodes. In other words, the record-type representation stores a node's type, any additional fields, and the child/parent information together in a single record, usually allocated as a unit. The pointers from one node to another node serve to link these records together. This makes it relatively memory efficient to load a single node in its entirety

into fast memory easily, but the allocator usually positions these nodes throughout the heap, making bulk or repeated loads of many nodes follow a random-access pattern.

Linearizing the node-edge information using a depth vector, all node structure sits in a single column in a matrix/table that contains the other node information in it as well. For simplicity, assume that every node has a single "extra" field and then a type field, giving the following tree using a single character for the type of the node and a string for the extra field value:



The matrix representation below is the same tree using a depth vector, which advantageously allocates the entire tree in a single contiguous region of memory, also contiguously allocating all sub-trees:

Depth	Type	Extra
0	T	n...
1	T	n...
2	T	n...
1	T	n...
2	T	n...
3	T	n...
2	T	n...
1	T	n...
2	T	n...
3	T	n...
3	T	n...
2	T	n...
3	T	n...
3	T	n...
2	T	n...

This contiguous allocation enables accessing whole chunks of the AST as large memory reads, but it may not permit bulk reads in the most efficient manner, as discussed in the following section.

3.2.4. Optimizing with Inverted Tables and Symbols

The matrix representation still does not quite satisfy the requirements for vectorization. It has a few serious faults when examined in the context of current array programming systems, particularly general-purpose systems that are not specialized. In a matrix representation of tables, each column has a different type. If the elements are simple, this results in a mixed array, but if the elements are nested, each element requires an internal array header. This leads to a great deal of inefficiency in the representation because each element must store type and/or shape information.

Also note that in many algorithms, processing the tree does not require the entire contents of the tree, only certain important aspects of it. A transformation may only require type and structural information while ignoring the rest of a tree's data. These are the first and second columns of a matrix representation. Often, a computation will only require a subset of the total available fields for the nodes over which it works. In this case, the matrix layout provides a better access pattern than the record-type representation, which is random access, because it enables a strided access pattern, assuming a row-major in memory representation, but this is still less efficient than the ideal sequential access pattern.

Some database applications and high-frequency/high-throughput time-series or financial applications address both the issue of mixed element types and column-wise access patterns using a technique known in the APL community as the inverted table, which optimizes the table/matrix

representation for cases where columns of the table are of the same type, but where each column is expected to be different. It replaces a matrix with a vector containing column vectors. This representation costs only a single array header for the table vector plus an array header per column, rather than array headers or type information on a per element basis. It provides a means for implementing column-major storage of row-major allocated and indexed tables of mixed field type in an efficient manner.

The inverted table representation enables independent computation over a single column with good control over the memory access and allocation. Patterns that would be random-access or strided-access in the previous representations are sequential access with an inverted table. Furthermore, it enables small element sizes for columns that do not require large numerical ranges. To further this advantage, the Co-dfns compiler assigns a named variable for each column vector in the table, rather than using a nested vector of vectors representation. The previous tree can be represented as an inverted table, depth vector representation as follows:

```
      d(15⍴'T')(↑15⍴⊂'n...')
```

0 1 2 1 2 3 2 1 2 3 3 2 3 3 2	TTTTTTTTTTTTTTT	n...
		n...
		n...
		n...
		n...
		n...
		n...
		n...
		n...
		n...
		n...
		n...
		n...
		n...
		n...

Instead of a 3-column matrix, a vector contains each column. Table (⍨) improves the visualization:

```
      ⍪¨d(15⍴'T')(↑15⍴⊂'n...')
```

0	T	n...
1	T	n...
2	T	n...
1	T	n...
2	T	n...
3	T	n...
2	T	n...
1	T	n...
2	T	n...
3	T	n...
3	T	n...
2	T	n...
3	T	n...
3	T	n...
2	T	n...

Examining the bytes required to represent the tree using each of the above techniques demonstrates the difference in efficiency using the Dyalog APL interpreter runtime:

```
      it←d(15⍴'T')(↑15⍴⊂'n...') ⍝ Inverted Table
      mt←((⍪d),'T'),⊂'Data'     ⍝ Matrix Representation
      ⎕SIZE¨'mt' 'it'
1960 256
```

Using depth vectors with an inverted table representation removes many of the inherent limitations of the record-type representation. However, the representation above still uses character vectors to represent the field data and types. Compiler implementations often rely heavily on atomic symbol as a core data element together with the language's record facilities for defining complex structures. Symbols represent atomic units of data with a string-like serialization compared primarily based on equality. These usually represent variable names and syntactic units within a compiler. Implementations often treat symbols like pointers, avoiding the need for string comparisons when

testing for equality. Furthermore, ASTs are usually constructed of a fixed set of specific record types. Record type systems tag the data representations to efficiently record the type of a given record, improving efficiency. Due to memory alignment restrictions, these tagging systems balance memory access performance and data compression, but the result is usually a reasonably efficient tagging system for objects. Combined with a symbol table, this avoids unnecessary duplication of string data such as is present in the above matrix or inverted table representation.

The type and extra field of the inverted table or matrix representation above are examples of object tagging and symbol data. The use of character vectors as above does not take advantage of the normal efficiency found with runtime tagged objects or symbol tables. The character vector representation can require up to 32-bits per character, depending on the size of character data in the system being used. Furthermore, the use of character arrays to represent symbolic/variable sorts of information in the tree will result in significant amounts of duplication for variable names.

Some APL implementations support general symbols but using them here may cost up to 64-bits per symbol or tag and limit computational flexibility over those columns. In the context of vector machines, care must be taken regarding this sort of limitation and space usage. Classical functional programming languages usually restrict the range of available computations to either an expensive string conversion or equality checking. Furthermore, the use of general symbols removes the freedom to control the data sizes in use.

In the Co-dfns compiler, the parser uses a manually controlled enumeration of types, dividing all nodes into a general type and sub-type called a kind using two columns of small range integers.

Because the number of types and kinds is very small, the system requires very few bits per element to represent these columns. Similarly, the parser generates an explicit symbol table in the form of a vector of unique character vectors and replaces all symbolic elements in the AST with negated indices to the appropriate symbol in the symbol table. Thus, all symbolic data in the AST changes to small range integer data. The compiler takes advantage of this representation to store multiple fields into a single column of the AST when they are mutually exclusive, and it also represents the symbolic data in the AST with as few bits as necessary. This saves considerable space in the representation. For this encoding, given a symbol table `ST`, a given symbol `N` is a negative integer whose string representation is `(|N)⊃ST`.

The above symbol internment optimizations result in an inverted table representation whose columns are likely to be very small range integers. This representation is very space efficient and better tuned to the access patterns most anticipated in the compiler.

To summarize, the parser produces an AST using the following techniques that it then passes to the compiler:

1. Linearizing the node-edge information of the tree into a depth vector
2. Storing node data as a table rather than separate node objects
3. Storing the table data as an inverted table instead of a row-major matrix
4. Interning string data using a separate symbol table
5. Storing node types as small integer enumerations
6. Storing each column as a bound variable

The AST as it comes from the parser consists of four columns:

Column	Name	Description
0	`d`	Depth vector of the AST
1	`t`	Primary class or Type of the nodes
2	`k`	Sub-class or Kind of the nodes
3	`n`	"Name" or "Reference" data for each node

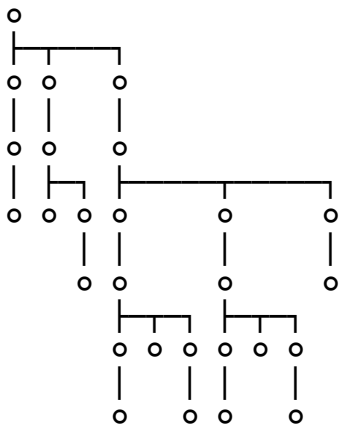
Columns 1 and 2 use small integer enumerations, while column 3 has its own, separate symbol table dedicated to its use. For example, consider the following namespace script:

```
:Namespace
 global←5
 gex←×global
 func←{X←⍵+global ⋄ Y←X×X ⋄ Y}
:EndNamespace
```

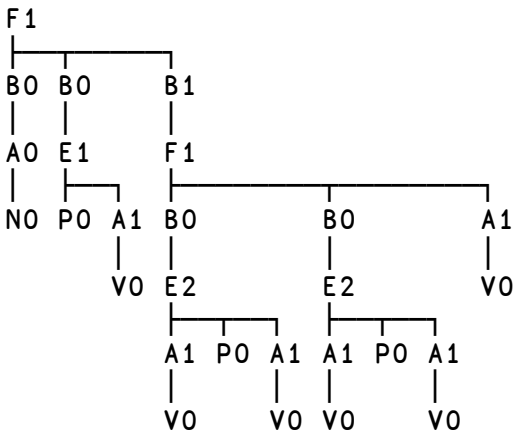
This structure mirrors the previous example tree relatively closely. The parser works as follows:

```
src←⊂':Namespace'
src,←⊂' global←5'
src,←⊂' gex←×global'
src,←⊂' func←{X←⍵+global ⋄ Y←X×X ⋄ Y}'
src,←⊂':EndNamespace'
ast←codfns.ps src
tree exports symbols←ast
```

The parser returns a tree that looks like this:



Annotating each node with its type and sub-kind gives the following:



The sub-kinds are numeric, while the types are symbols interned with the following symbol vector:

```
      NΔ
ABEFGLMNOPVZ
```

Notice that the parser returns the tree, a table of the exported names, and the symbol table for the `n` field. Visualizing these results gives the following, using the Table primitive to beautify the results:

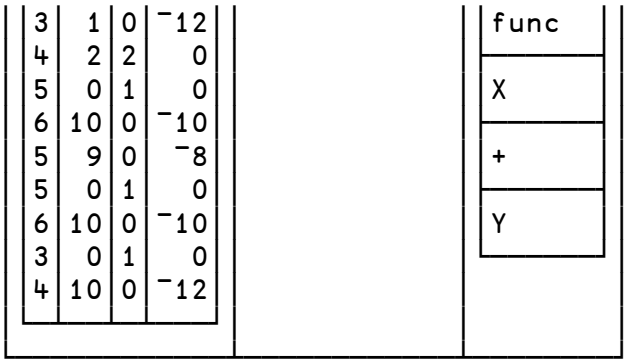
```
      (⍪¨tree)(⍪¨exports)(⍪symbols)
```

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

func	1
gex	0
global	0

0
ω
α
αα
ωω
global
5
gex
×

```
...
```



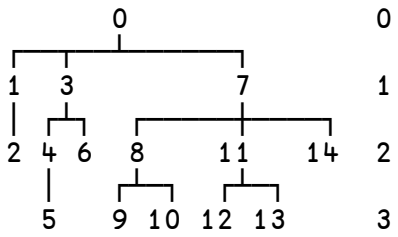
The following discussion of encoding ASTs assumes an interned symbol `n` field and similarly encoded type `t` and kind `k` fields. It also assumes that the AST is represented as an inverted table, as above.

The depth vector representation is space efficient and can be constructed easily while parsing. However, as discussed previously, it is not ideal for manipulating the tree, because manipulation may incur heavy costs to reorder nodes in the AST, to add nodes, and to determine the critical parent-child relationships when working with the AST. The following section removes the ordering dependencies from the depth vector using a path matrix, and the section after that discusses how to compress this concept as a parent vector.

3.2.5. Path Matrices and Decoupling Table Ordering and Node Edges

Decoupling of the ordering dependency of the depth vector representation means enabling local reasoning about the specific edges of nodes in, ideally, a constant time manner. The strongest requirement for such independence states that any two arbitrary nodes in the tree may be compared efficiently and their relative locations in the tree totally understood. One way to do this is a path matrix.

A path is a vector of node ids listing the ancestors of a given node up to and including the root node. A path matrix is a matrix where one dimension is the size of the elements in the tree, and the other is the size of the maximum depth in the tree, that is, the longest path in the tree from the root node to one of the leaf nodes. Recall the simple tree given above along with its depths and node ids:



The corresponding path matrix to this tree is a matrix of shape `4 15`, represented as follows:

```
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
   1  1  3  3  3  3  7  7  7  7  7  7  7  7
      2     4  4  6     8  8  8 11 11 11 14
               5              9 10    12 13
```

The above matrix omits the "null" spaces to facilitate a better visualization of the path matrix. Notice that each column of the matrix is a path from a given node id to the root node and thus the first row of all 0's since the tree has only a single root node.

Consider the construction of the path matrix for the example namespace script parsed above. The node count of the tree is 27. Recall the depth vector of the tree:

```
      d
0 1 2 3 1 2 3 3 4 1 2 3 4 5 6 5 5 6 3 4 5 6 5 5 6 3 4
```

The depth vector gives positions for each node in accordance with their depths in a matrix of shape `7 27` (that is, the depth and the node count of the tree). Starting with a matrix of empty spaces gives the following visualization:

```
      ⍳≠d
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26
      ⍉↑d,¨⍳≠d
0 1 2 3 1 2 3 3 4 1  2  3  4  5  6  5  5  6  3  4  5  6  5  5  6  3  4
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26
      (⍳≠d)@(d,¨⍳≠d)⊢7 27⍴' '
0
  1     4          9
    2     5          10
      3     6 7        11                    18                    25
                8          12                    19                    26
                              13    15 16           20    22 23
                                 14        17           21        24
```

This provides a transposed tree-view layout of the node ids, much like certain file management applications use to display a tree. Indeed, transposing the above layout provides a familiar nested representation:

```
       ⍉(⍳≠d)@(d,¨⍳≠d)⊢7 27⍴' '
  0
    1
       2
          3
    4
       5
          6
          7
             8
    9
      10
         11
            12
               13
                  14
               15
               16
                  17
         18
            19
               20
                  21
               22
               23
                  24
         25
            26
```

The above matrix utilizes a combination of character and numeric data that is not suitable for computation. Using a zero-filled array instead gives this:

```
      (⍳≠d)@(d,¨⍳≠d)⊢7 27⍴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 0 4 0 0 0 0 9  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0 0 2 0 0 5 0 0 0 0 10  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0 0 0 3 0 0 6 7 0 0  0 11  0  0  0  0  0  0 18  0  0  0  0  0  0 25  0
0 0 0 0 0 0 0 0 8 0  0  0 12  0  0  0  0  0  0 19  0  0  0  0  0  0 26
0 0 0 0 0 0 0 0 0 0  0  0  0 13  0 15 16  0  0  0 20  0 22 23  0  0  0
0 0 0 0 0 0 0 0 0 0  0  0  0  0 14  0  0 17  0  0  0 21  0  0 24  0  0
```

This is the same structure as above, but with the spaces filled in with zeroes. Using the node id as the "fill element" for all rows in a column greater than the length of the path allows a combination of max scans to generate the following path matrix:

```
      ⊢PM←⌈\⌈⍀(⍳≠d)@(d,¨⍳≠d)⊢7 27⍴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 1 1 4 4 4 4 4 9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9
0 1 2 2 4 5 5 5 5 9 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10
0 1 2 3 4 5 6 7 7 9 10 11 11 11 11 11 11 11 18 18 18 18 18 18 18 25 25
0 1 2 3 4 5 6 7 8 9 10 11 12 12 12 12 12 12 18 19 19 19 19 19 19 25 26
0 1 2 3 4 5 6 7 8 9 10 11 12 13 13 15 16 16 18 19 20 20 22 23 23 25 26
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26
```

Note how the node id for each column extends down the column to fill all the remaining space in the column beyond the depth of the node.

The path matrix representation, which harkens back to Iverson's APL book, has the advantage that each column can now serve as a fully qualified, unique identifier for a given node. Additionally, these qualifiers are comparable, meaning that each of them encodes enough information in them to compare any two arbitrary columns without reference to the rest of the matrix. Assuming row-major allocation, the transposition of this matrix enables contiguous access to each column (now row), giving good data locality.

As an example of working with path matrices, consider the problem of computing the distance of each node from every other node in the tree. The distance of a node from another node is the minimal number of nodes visited along the tree in order to traverse from one node to another. One way to think of this is that the maximum potential distance is the sum of the corresponding depths plus 2 (to account for the zero-indexing of the depth vector). The actual distance, therefore, is the maximum potential distance subtracted by twice the depth plus 1 of the nearest common ancestors of the nodes. Because the node id is a unique fill element for each column in the path matrix, the APL idiom `+.=` computes the length of the common prefix shared between two paths, which is the depth plus 1 of the nearest common ancestors. The following expression computes a distance matrix for the tree described by `d` and `PM` above:

```
      0⌈(∘.+⍨1+d)-2×(⍉+.=⊢)PM
0 1 2 3 1 2 3 3  4 1 2 3 4 5  6 5 5  6 3 4 5  6 5 5  6 3 4
1 0 1 2 2 3 4 4  5 2 3 4 5 6  7 6 6  7 4 5 6  7 6 6  7 4 5
2 1 0 1 3 4 5 5  6 3 4 5 6 7  8 7 7  8 5 6 7  8 7 7  8 5 6
3 2 1 0 4 5 6 6  7 4 5 6 7 8  9 8 8  9 6 7 8  9 8 8  9 6 7
1 2 3 4 0 1 2 2  3 2 3 4 5 6  7 6 6  7 4 5 6  7 6 6  7 4 5
2 3 4 5 1 0 1 1  2 3 4 5 6 7  8 7 7  8 5 6 7  8 7 7  8 5 6
3 4 5 6 2 1 0 2  3 4 5 6 7 8  9 8 8  9 6 7 8  9 8 8  9 6 7
3 4 5 6 2 1 2 0  1 4 5 6 7 8  9 8 8  9 6 7 8  9 8 8  9 6 7
4 5 6 7 3 2 3 1  0 5 6 7 8 9 10 9 9 10 7 8 9 10 9 9 10 7 8
1 2 3 4 2 3 4 4  5 0 1 2 3 4  5 4 4  5 2 3 4  5 4 4  5 2 3
2 3 4 5 3 4 5 5  6 1 0 1 2 3  4 3 3  4 1 2 3  4 3 3  4 1 2
3 4 5 6 4 5 6 6  7 2 1 0 1 2  3 2 2  3 2 3 4  5 4 4  5 2 3
4 5 6 7 5 6 7 7  8 3 2 1 0 1  2 1 1  2 3 4 5  6 5 5  6 3 4
5 6 7 8 6 7 8 8  9 4 3 2 1 0  1 2 2  3 4 5 6  7 6 6  7 4 5
6 7 8 9 7 8 9 9 10 5 4 3 2 1  0 3 3  4 5 6 7  8 7 7  8 5 6
5 6 7 8 6 7 8 8  9 4 3 2 1 2  3 0 2  3 4 5 6  7 6 6  7 4 5
5 6 7 8 6 7 8 8  9 4 3 2 1 2  3 2 0  1 4 5 6  7 6 6  7 4 5
6 7 8 9 7 8 9 9 10 5 4 3 2 3  4 3 1  0 5 6 7  8 7 7  8 5 6
3 4 5 6 4 5 6 6  7 2 1 2 3 4  5 4 4  5 0 1 2  3 2 2  3 2 3
4 5 6 7 5 6 7 7  8 3 2 3 4 5  6 5 5  6 1 0 1  2 1 1  2 3 4
5 6 7 8 6 7 8 8  9 4 3 4 5 6  7 6 6  7 2 1 0  1 2 2  3 4 5
6 7 8 9 7 8 9 9 10 5 4 5 6 7  8 7 7  8 3 2 1  0 3 3  4 5 6
...
```

```
5 6 7 8 6 7 8 8  9 4 3 4 5 6  7 6 6  7 2 1 2  3 0 2  3 4 5
5 6 7 8 6 7 8 8  9 4 3 4 5 6  7 6 6  7 2 1 2  3 2 0  1 4 5
6 7 8 9 7 8 9 9 10 5 4 5 6 7  8 7 7  8 3 2 3  4 3 1  0 5 6
3 4 5 6 4 5 6 6  7 2 1 2 3 4  5 4 4  5 2 3 4  5 4 4  5 0 1
4 5 6 7 5 6 7 7  8 3 2 3 4 5  6 5 5  6 3 4 5  6 5 5  6 1 0
```

The path matrix representation of a tree does solve the dependency issue of the depth vector, but at the cost of `N×D` memory, where `N` is the cardinality of the tree and `D` the depth of the tree. This results in excessive memory usage as tree size begins to grow. Two major benefits of the path matrix representation are the true independence that the representation gives in assessing the relationship between arbitrary nodes in the tree, and the data locality benefits of storing each node's full path in a localized manner per node. Parent vectors provide an alternative that avoids the memory costs.

3.2.6. Parent Vector Representation

Most tree manipulations do not require the full comparative power provided by a path matrix. Instead, common manipulations tend to focus on walking up and down the tree, or left and right along the tree, rather than arbitrary walks throughout the whole tree. Thus, for any given node, its ancestors or immediate siblings, rather than, say, cousins matter more. A parent vector avoids duplication of data and requires only linear memory. It optimizes for ancestor or sibling traversals rather than arbitrary node-node comparisons. This optimization saves on memory usage while providing good asymptotic performance for the most important tree traversals in compiler passes.

The parent vector representation uses a pointer vector encoding the parent relationships in the tree. (It is also possible to use a pointer vector to represent siblings in the same way, but this work does not utilize this technique.) A pointer vector is an index vector whose elements index into itself.

That is, for a given pointer vector `V` the assertion `∧/V∈⍳≢V` holds true. In a parent vector each element `p[i]` points to the parent id of node `i` in `p`. When node `i` is a root node, then `p[i]=i`. That is, root nodes point to themselves in the parent vector.

The parent vector partially orders the tree, so that the order of siblings matters in the vector, but nodes may otherwise appear in any order within the vector. The introduction of a sibling vector applies a total ordering to the tree, enabling full decoupling at the cost of maintaining two pointer vectors. Having a partially ordered, rather than fully ordered vector enables higher performance operations when adding or deleting nodes than might otherwise be possible.

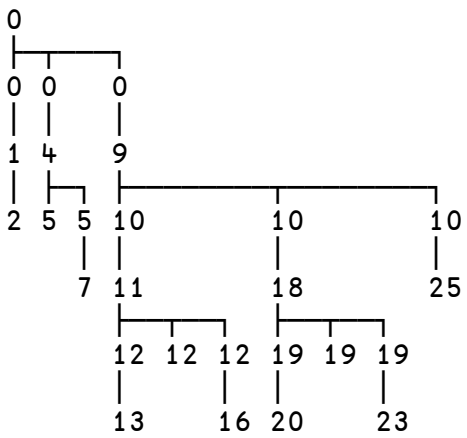
Repeatedly indexing into the parent vector visits the ancestors of a node in order up the tree. Thus, the parents of all nodes are `p`, the grandparents of all nodes are `p[p]`, and so on. The root node of all elements in the tree is given by the fixpoint `I⍣≡⍨p` where `I←{(⊂⍵)[]⍺}`. Note that the parent vector, path matrix, and depth vector representations all permit multiple disconnected trees within a single AST structure.

The guarantee that `∧/p∈⍳≢p`, that is, that all elements of `p` fall within the index range of `p`, combined with the fact that the AST does not have cycles, allows root nodes to point to themselves unambiguously, rather than using a null value as often found in HPC applications that use similar structures. The use of a null value often necessitates branching to check ranges before doing the main work, potentially introducing divergence and reducing performance. Self-reference ensures all elements are valid indices of `p`.

Recall the previous depth vector example:

```
        d
0 1 2 3 1 2 3 3 4 1 2 3 4 5 6 5 5 6 3 4 5 6 5 5 6 3 4
```

Here is the same tree with the parents visualized graphically:



This gives the following parent vector:

```
       p
0 0 1 2 0 4 5 5 7 0 9 10 11 12 13 12 12 16 10 18 19 20 19 19 23 10 25
```

Note that the depth-first, pre-order traversal given above matches the order of the depth vector, but this is not required, and any other permutation that preserved sibling order would work provided that the values pointed to the appropriate elements. Many tree algorithms do not require ordered siblings, in which case the parent vector has no ordering requirements at all.

The parent vector is relatively space efficient, less so than the depth vector representation and significantly more so than the path matrix representation. It is also very efficient to compute over in general and relatively easy to optimize for specific access patterns. While it does not enable independent comparisons between distantly related nodes as does the path matrix representation, it allows for critical paths significantly better than the ordered depth vector representation in common

cases. It strikes a balance between space efficiency and computational performance, and indeed, as shall be seen in the following sections, space efficiency is not independent from run time performance.

In addition to computational benefits, the parent vector provides the easiest means by which to add new nodes. Often, new nodes may be added by appending them to the vector without additional reordering. In cases where appending is not possible, the partial ordering enables more flexibility when inserting nodes that improves overall performance. Compiler algorithms involve significant node addition so this advantage should not be under-estimated. Deleting nodes in the parent vector might appear more difficult compared to a depth vector, but it scarcely requires more code in practice, as discussed below.

Finally, changing edges in the tree using parent vectors exhibits good memory behavior and essentially no copying overhead. Many key algorithms admit efficient access patterns with low constant factors, low over computation, and no excessive memory requirements, which allows for competitive implementations even at small data sizes.

Using the above `p` vector, together with the previous `t`, `k`, and `n` vectors, gives the following AST in parent vector, inverted table format:

```
      ;¨p  t  k  n
┌──┬──┬──┬──┐
│ 0│ 3│ 1│  0│
│ 0│ 1│ 0│ ¯5│
│ 1│ 0│ 0│  0│
│ 2│ 7│ 0│ ¯6│
│ 0│ 1│ 0│ ¯7│
│ 4│ 2│ 1│  0│
│ 5│ 9│ 0│ ¯8│
│ 5│ 0│ 1│  0│
│ 7│10│ 0│ ¯5│
...
```

```
 0  1 1 ¯10
 9  3 1   0
10  1 0 ¯11
11  2 2   0
12  0 1   0
13 10 0  ¯1
12  9 0 ¯12
12  0 1   0
16 10 0  ¯5
10  1 0 ¯14
18  2 2   0
19  0 1   0
20 10 0 ¯11
19  9 0  ¯8
19  0 1   0
23 10 0 ¯11
10  0 1   0
25 10 0 ¯14
```

3.3. Converting from Depth Vector to Parent Vector

The parser encodes the structure of the AST in a depth vector, but the compiler works over a parent vector representation, first converting from the depth vector `d` to a parent vector `p`. Recall that `d[i]` for node `i` in depth vector `d` gives the depth of that node in the tree, while `p[i]` gives an index into `p` that corresponds to the parent of node `i`. Note here the convention of referring to a node as an index into one of these "structure" vectors, such as a depth vector or a parent vector. Thus, the expression `p[i]` is said to give the parent node of node `i`, it being an index is an implicit part of the definition.

The depth vector encodes the parent-child information. More formally, the parent node of node `i` in depth vector `d` is a node `j` where `d[j]=d[i]-1` and `j<i` such that `i-j` is the smallest possible value among candidates for `j`. Another way of phrasing this is to say that the parent node of a given node is the nearest node to the left whose value is one less in depth. Assuming a single tree with one root, the expression `⊃⌽⍳d[⍳i]=d[i]-1` calculates the parent node. To see this working,

recall the previous example depth vector and its associated tree giving the id of each node in the tree and the associated depth:

```
      d
0 1 2 1 2 3 2 1 2 3 3 2 3 3 2

      0          0
┌──┬──┴───────┐
1  3          7      1
│ ┌┴┐  ┌──────┼────┐
2 4 6  8     11   14 2
  │   ┌┴─┐  ┌┴──┐
  5   9 10 12 13     3
```

To find the parent of node 14, which is 7, the above expression decomposes into the following pieces:

```
      d←0 1 2 1 2 3 2 1 2 3 3 2 3 3 2
      i←14
      d[i]-1
1
      ⍳i
0 1 2 3 4 5 6 7 8 9 10 11 12 13
      d[⍳i]
0 1 2 1 2 3 2 1 2 3 3 2 3 3
      d[⍳i]=d[i]-1
0 1 0 1 0 0 0 1 0 0 0 0 0 0
      ⍸d[⍳i]=d[i]-1
1 3 7
      ⌽⍸d[⍳i]=d[i]-1
7 3 1
      ⊃⌽⍸d[⍳i]=d[i]-1
7
```

This expression is simple and intuitively matches the formal idea of finding a parent of a single node, but it does not scale elegantly over all nodes in `d` to convert the entire vector `d` into a parent vector. An improvement removes the use of the final `⊃⌽⍸` idiom for something that readily scales to multiple nodes at a time. The expression `¯1++/∨\⌽d[⍳i]=d[i]-1` computes the same thing but using the common APL pattern of summing an OR-scan to compute the index of the first non-zero element in a vector. The breakdown is as follows:

```
      d[⍳i]=d[i]-1
0 1 0 1 0 0 0 1 0 0 0 0 0 0
      ∨\⍲d[⍳i]=d[i]-1
0 0 0 0 0 0 1 1 1 1 1 1 1 1
      +/∨\⍲d[⍳i]=d[i]-1
8
      ¯1++/∨\⍲d[⍳i]=d[i]-1
7
```

Critically, the OR-scan technique does not just work on vectors, but also works to compute the column index of the first non-zero element for each row in a matrix. This permits a new, more general expression in place of the expression `d[⍳i]=d[i]-1` that produces a matrix of such Boolean rows, each corresponding to one element in `d` to which applies the OR-scan idiom to compute the appropriate parent for each element. To do this, first note that the expression `∘.>⍤¨d` gives a Boolean matrix where index `(i,j)` is a one if node `i` appears deeper in the tree (that is, has a greater depth) than node `j`. Thus, each row of the matrix is a potential candidate parent.

```
      ∘.>⍤¨d
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
1 1 0 1 0 0 0 1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 0 1 0 0 0 1 0 0 0 0 0 0 0
1 1 1 1 1 0 1 1 1 0 0 1 0 0 1
1 1 0 1 0 0 0 1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 0 1 0 0 0 1 0 0 0 0 0 0 0
1 1 1 1 1 0 1 1 1 0 0 1 0 0 1
1 1 1 1 1 0 1 1 1 0 0 1 0 0 1
1 1 0 1 0 0 0 1 0 0 0 0 0 0 0
1 1 1 1 1 0 1 1 1 0 0 1 0 0 1
1 1 1 1 1 0 1 1 1 0 0 1 0 0 1
1 1 0 1 0 0 0 1 0 0 0 0 0 0 0
```

This gives the same Boolean vector for each row as `d[⍳i]=d[i]-1` except the upper right triangle portion. The OR-scan idiom requires a zeroed upper right triangle in the matrix. The expression `∘.>⍨⍳≠d` produces a lower left triangular mask:

```
      ∘.>⍨⍳≠d
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
1 1 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 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
1 1 1 1 1 1 1 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1 1 0 0 0 0
1 1 1 1 1 1 1 1 1 1 1 1 0 0 0
1 1 1 1 1 1 1 1 1 1 1 1 1 0 0
1 1 1 1 1 1 1 1 1 1 1 1 1 1 0
```

Applying AND over the two matrices produces an appropriate replacement for the original `d[⍳i]=d[i]-1` expression that represents the same data for all nodes in `d`:

```
      (∘.>⍨d)∧∘.>⍨⍳≠d
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
1 1 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
1 1 0 1 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 0 0 0 0 0 0 0 0 0 0
1 1 0 1 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
1 1 0 1 0 0 0 1 0 0 0 0 0 0 0
1 1 1 1 1 0 1 1 1 0 0 0 0 0 0
1 1 1 1 1 0 1 1 1 0 0 0 0 0 0
1 1 0 1 0 0 0 1 0 0 0 0 0 0 0
1 1 1 1 1 0 1 1 1 0 0 1 0 0 0
1 1 1 1 1 0 1 1 1 0 0 1 0 0 0
1 1 0 1 0 0 0 1 0 0 0 0 0 0 0
```

This expression can then be composed with the OR-scan sum idiom to compute the appropriate parent vector from a depth vector, which breaks down as follows:

```
       ϕ(∘.>⍨d)∧∘.>⍨⍳≠d
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 0 0 0 0 0 0 0 0 0 0 0 0 1 1
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 1 0 1 1
0 0 0 0 0 0 0 0 0 0 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0 0 1 0 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 1 0 0 0 1 0 1 1
0 0 0 0 0 0 1 1 1 0 1 1 1 1 1
0 0 0 0 0 0 1 1 1 0 1 1 1 1 1
0 0 0 0 0 0 0 1 0 0 0 1 0 1 1
0 0 0 1 0 0 1 1 1 0 1 1 1 1 1
0 0 0 1 0 0 1 1 1 0 1 1 1 1 1
0 0 0 0 0 0 0 1 0 0 0 1 0 1 1
       ⍲\ϕ(∘.>⍨d)∧∘.>⍨⍳≠d
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 0 0 0 0 0 0 0 0 0 0 0 0 1 1
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 1 1 1 1
0 0 0 0 0 0 0 0 0 0 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0 0 1 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 1 1 1 1 1 1 1 1
0 0 0 0 0 0 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 1 1 1 1 1 1 1 1
0 0 0 1 1 1 1 1 1 1 1 1 1 1 1
0 0 0 1 1 1 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 1 1 1 1 1 1 1 1
       +/⍲\ϕ(∘.>⍨d)∧∘.>⍨⍳≠d
0 1 2 1 4 5 4 1 8 9 9 8 12 12 8
       ¯1++/⍲\ϕ(∘.>⍨d)∧∘.>⍨⍳≠d
¯1 0 1 0 3 4 3 0 7 8 8 7 11 11 7
       0⌈¯1++/⍲\ϕ(∘.>⍨d)∧∘.>⍨⍳≠d
0 0 1 0 3 4 3 0 7 8 8 7 11 11 7
```

The above final expression relatively directly expressions the intuition of finding the parent for a given node by finding the nearest leftwards node whose depth is less than the given node's depth. Unfortunately, the runtime complexity of this expression is quadratic in both space and time,

dominated by the outer product calculations. The critical path for the computation is linear. This presents an unfortunate choice to either trade space usage for time or vice versa. Moreover, the linear critical path begs for improvement. Fortunately, better solutions exist, while still utilizing the same basic intuition, but reducing the size of the data requirements and improving the asymptotic behavior.

To make these improvements, observe a few properties. Firstly, the depth of a node's parent is exactly 1 less than its depth. It then follows that at least one element of each member of the set `⍳d[i]` must appear in `d` before node `i`, that is, in the vector `d[⍳i]`. The expression `f⌸v` groups common elements of vector `v` and applies function `f` once for each unique element in `v` where the left argument to `f` is the unique element (the key) and the right argument is a vector of the indices into `v` whose values are all equal to the key. The result is an array with as many major cells as unique elements in `v`. Thus, the expression `⊢∘⊂⌸d` gives a vector (nested) of vectors of node IDs, where the `i`th element is a vector of all nodes of depth `i`, in depth-first pre-order traversal ordering. In short, it groups all nodes by their depth, sorted in ascending order by depth. This expression restructures the view into the depth vector in a critical way to enable a more efficient search for parent nodes.

The quadratic computation of `p` suffers from over-work, since it compares every node against every other node twice. Really, for a node of depth `d`, the parent must have depth `d-1`. With nodes grouped by depth as above, the parents of the nodes in group `i>0` must be in group `i-1`. In the case of group 0, they have no parents but themselves. This suggests an improved implementation. Begin by assuming all nodes are root nodes by letting `p←⍳≢d`, then use the above grouping to fix up `p`. First examine the result of `⊢∘⊂⌸d`:

```
      ⊢∘⊂⌸d
┌─┬─────┬─────────────┬──────────────┐
│0│1 3 7│2 4 6 8 11 14│5 9 10 12 13  │
└─┴─────┴─────────────┴──────────────┘
```

Now consider groups 2 and 3, corresponding to the nodes whose depths are 2 and 3, respectively. All of group 3's parents appear in group 2. Note that for each element in group 3, there is an interval where that element can "squeeze" between two adjacent elements in group 2 while maintaining the ordering. Intuitively, every node in a depth vector appears in the depth vector between its parent and either the end of the vector or a parent's sibling or ancestor. That is, a node and its right sibling or nearest right ancestor form an interval of space in which all the node's descendants appear. The grouping above makes these intervals explicit. It does not matter if the intervals are slightly larger in width than strictly necessary, because it does not change the left element of the interval. The existence of these intervals changes the process of finding the parent node into a problem of identifying the corresponding interval into which a given node belongs. This computation exactly corresponds to the Interval Index operation ⍸, which returns an index into its left argument for each element in its right that points to the interval described in the left argument to which that element belongs. Taking groups 2 and 3 from above, ⍸ gives the following result with group 2 as the left argument and group 3 as the right:

```
      2 4 6 8 11 14⍸5 9 10 12 13
1 3 3 4 4
```

Using this result to index into group 2 gives the appropriate parents for group 3:

```
      a←2 4 6 8 11 14
      w←5 9 10 12 13
      a[a⍸w]
4 8 8 11 11
```

To compute the parents, then, one need only to apply the above computation to each pair of adjacent groups that appear in the grouping given by `⊢∘⊂⌸d`. Using the 2-wise reduction operator to apply this function gives the following expression for computing the corresponding parent vector from a depth vector `d`:

```
      p⍪2{p[⍵]←⍺[⍺⍸⍵]}⌿⊢∘⊂⌸d⍪p←⍳≢d
0 0 1 0 3 4 3 0 7 8 8 7 11 11 7
```

Note that the computational overhead of this code is much reduced from the previous computation. The reduction causes `≢d` writes to `p`, and because `⍺` and `⍵` are guaranteed to be sorted, the `⍺[⍺⍸⍵]` is a linear operation. This makes the reduction itself linear. The `⊢∘⊂⌸` is fundamentally a sort over small range integers, and thus linear as well. This gives `O(≢d)` space and time complexity on a serial machine. For parallel machines, note that the writes to `p` are independent, allowing the reduction to occur in parallel. A binary search based log-linear implementation of `⍸` using standard techniques results in an `O(⊛≢⍺)` critical path. The critical path of `⊢∘⊂⌸` is `O(⊛≢d)`. The total critical path is thus `O((⊛m)+⊛≢d)` where `m` is the maximum branch size of the AST. In the case of space, a fused implementation of `⍸` will use linear space.

This improved implementation is significantly more efficient than the reference implementation in time and space on both sequential and parallel machines, while also being simpler/shorter, if slightly less obvious.

3.4. Computing the Nearest Lexical Contour

After converting the depth vector into a parent vector, the compiler records the lexically nearest function node that encloses each node in the AST. That is, it annotates each node in the AST with a new field `r` that records the function node (recall that a node is just a pointer/index to its field data in the inverted table representation of the AST) that is the most immediate ancestor of function class/type. In the dfns syntax, there is a one to one mapping between lexical contours/scopes and user-defined functions and namespaces. This pass groups or colors nodes by their lexical contour/scope to use in future passes. The compiler stores this information in a new field/column vector `r` whose length is equal to `p` and whose values are all pointers/indexes into `p`. The computation itself amounts to a relatively straightforward walk through the tree to connect each node with its appropriate lexical reference node. As such, it is the simplest demonstration of the fundamental concept of traversing an AST in the compiler.

In a traditional representation, such a tree walk begins at the root of the tree and recursively descends deeper into the tree. In this case, a state variable usually maintains the nearest function node, and the pass annotates each node while traversing down the tree; each time the pass encounters a function node it updates the state variable while traversing through that function's descendants. The following recursive function illustrates this pattern. The left argument `⍺` maintains the current nearest lexical node (function node) while the right argument `⍵` is the current tree to annotate. It assumes an accessor for a node's id and for its kids, two predicates for testing leaf-ness and whether a node is a lexical node, as well as constructors for leaf nodes and branches.

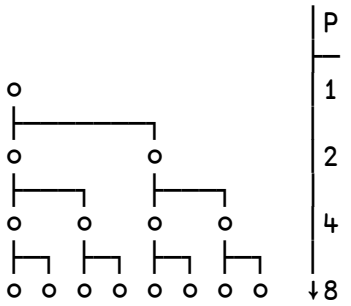
```
NearestScope←{α←id ω
    isΔleaf ω : makeΔleaf α
    isΔF ω    : α makeΔtree (id ω) ∇ kids ω
    α makeΔtree α ∇ kids ω}
```

This approach benefits from using the call stack and explicit variables to track information through the tree in a single-threaded manner. In the above example, only a single variable suffices to recall the nearest lexical node since it relies on the call stack maintain the proper value while moving along the tree. Because the computation is single threaded and the use of pointers in the language allows for information to be readily distributed and replicated to different parts of the call stack without consuming additional memory, the results on single-threaded machines tend to perform reasonably well, and have good computational and space complexity if designed correctly.

In the context of parallel machines, this top-down approach suffers some challenges. The parallel updating of complex information structures, such as lookup tables or the like, may suffer from either duplication or synchronization overheads as more information needs to be propagated further through the tree. And perhaps more fundamentally, traversing a tree dynamically from the top down requires the ability to dynamically adjust parallelism to either increase or reduce parallelism as the tree-walk proceeds. This leads to irregular parallelism patterns that are difficult or impossible to predict before the beginning of the execution absent a specific tree against which to optimize.

Choosing layouts that optimize for one type of traversal or another in top-down fashion mitigates this issue for certain use cases, but lack generality because they bias traversals towards patterns that may not suit a specific problem. While they can improve parallelism, they don't

necessarily eliminate all the issues with a top-down traversal. The following diagram illustrates how the top down approach exposes increasing parallelism as one traverses deeper down the tree:

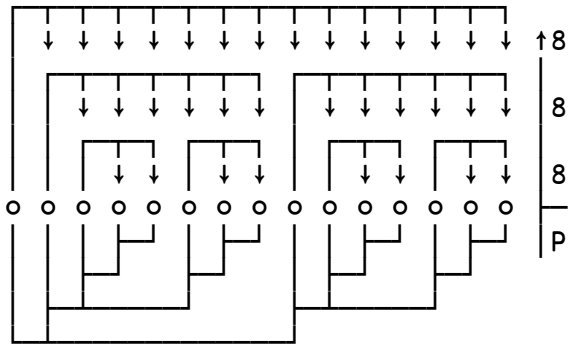


At each level of the traversal, branching exposes further opportunities for parallelism, and it is not possible to know in advance the degree of parallelism to choose without examining the tree. With course-grained parallelism on SMP machines, work-stealing schedulers have proven effective in handling dynamic parallelism of this sort. However, they are somewhat complicated to implement, and can introduce significant engineering costs into the code to make them work well. On fine-grained parallel systems, it becomes much more important to know the amount of parallelism in advance, since, for instance, systems like CUDA perform best when they can fully utilize the provided SIMD features.

In contrast, the parent vector representation inverts the standard record-type representation of parents and children. Rather than each parent node pointing to a list of its children, each child maintains a single pointer to its parent. This representation sees significant use in some computing domains, but it does not see common use when designing tree manipulation algorithms, which all tend to emphasize top-down, recursive descent traversals. The parent vector biases traversal from a

top-down direction to a bottom-up one. For each node in the tree, it is relatively easy to walk up the parent vector to the root node, but less obvious how to traverse down the tree.

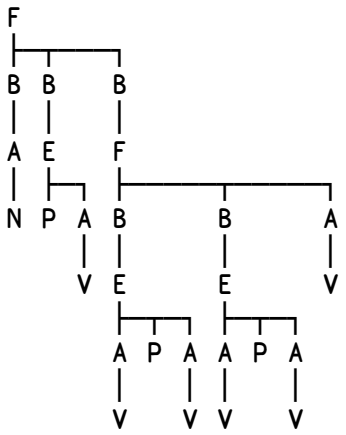
Fine-grained SIMD parallelism greatly favors this opposite traversal direction, and the Co-dfns compiler traverses up the tree when doing such tree-walks. The compiler computes in parallel over every node in the AST, starting with maximum parallelism using the parent vector to walk upwards along the tree to find the desired information. The following diagram describes the traversal:



The arrows indicate parent information propagated to various vector lanes or threads. Note the constant number of parallel threads working in lockstep. This makes the computation of the next step significantly simpler as each thread individually traverses up the tree to the root and never requires synchronization with any of the other threads.

The above diagram explains the theoretical model. In practice, both approaches pose engineering challenges, but overcoming with these challenges is simpler and less involved with the bottom up approach than the top-down one.

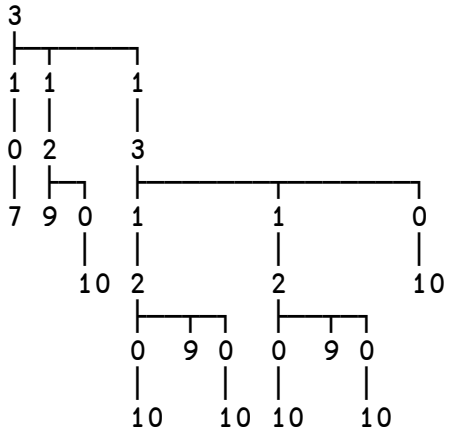
Computing the nearest lexical contour demonstrates a simple bottom up traversal. Recall the sample AST from previous sections:



In the above tree, the `p` and `t` vectors are as follows:

```
      p
0 0 1 2 0 4 5 5 7 0 9 10 11 12 13 12 12 16 10 18 19 20 19 19 23 10 25
      t
3 1 0 7 1 2 9 0 10 1 3 1 2 0 10 9 0 10 1 2 0 10 9 0 10 0 10
```

Rendering the above tree using the type vector directly instead of the symbolic representation of each type yields the following:



In this encoding the nodes of type 3 are lexical nodes. Like the previous recursive snippet, the pass retains only the node idea of the nearest lexical node. More complex passes could require a more sophisticated auxiliary structure than just the id. This pass extracts the node id of the nearest enclosing node of type 3, as demonstrated in the following exposition:

```
      p
0 0 1 2 0 4 5 5 7 0 9 10 11 12 13 12 12 16 10 18 19 20 19 19 23 10 25
      t
3 1 0 7 1 2 9 0 10 1 3 1 2 0 10 9 0 10 1 2 0 10 9 0 10 0 10
      t[p]
3 3 1 0 3 1 2 2 0 3 1 3 1 2 0 2 2 0 3 1 2 0 2 2 0 3 0
      t[p]≠3
0 0 1 1 0 1 1 1 1 0 1 0 1 1 1 1 1 1 0 1 1 1 1 1 1 0 1
      (t[p]≠3)/p
1 2 4 5 5 7 9 11 12 13 12 12 16 18 19 20 19 19 23 25
      p I (t[p]≠3)/p
0 1 0 4 4 5 0 10 11 12 11 11 12 10 18 19 18 18 19 10
      p I@{t[⍵]≠3} p
0 0 0 1 0 0 4 4 5 0 0 10 10 11 12 11 11 12 10 10 18 19 18 18 19 10 10
      p I@{t[⍵]≠3} p I@{t[⍵]≠3} p
0 0 0 0 0 0 0 0 4 0 0 10 10 10 11 10 10 11 10 10 10 18 10 10 18 10 10
      p I@{t[⍵]≠3} p I@{t[⍵]≠3} p I@{t[⍵]≠3} p
0 0 0 0 0 0 0 0 0 0 0 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10
      p I@{t[⍵]≠3}⍤≡ p
0 0 0 0 0 0 0 0 0 0 0 0 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10
      I@{t[⍵]≠3}⍤≡⍨p
0 0 0 0 0 0 0 0 0 0 0 0 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10
```

Given a suitable parent vector `p` and type vector `t`, this expression returns the node id for the nearest enclosing scope for each node in the AST. Recall that `p[⍵]` gives a vector containing the parents of nodes `⍵`. For nodes `p[⍵]` of function type (type 3), nothing need be done, having found the nearest enclosing function. However, for nodes not of function type, the traversal must examine the parent of `p[⍵]`, and so forth until finding a function node. The pass continuously indexes on `p` until finding the function nodes for all the initial nodes, determined by finding the fixed point.

In short, it traverses the parent vector `p` for each node until finding a node that is of type 3. This idiom `I@B⍤≡` is the traversal idiom. More formally, for a pointer vector left argument `⍺` and a set of node ids `⍵` as the right argument, the above idiom will replace element `i` in `⍵` with `⍺[⍵[i]]` so long as `(B ⍵)[i]` gives `1`, repeating this to a fixed point. It enables traversing and extracting

information about the tree that is relevant for each compiler pass on a per node basis, in a data-parallel fashion. It is the primary "traversal" idiom used when a pass must traverse the structure of the tree.

The critical path of each iteration is constant, since it uses only fully SIMD parallel operations. The iteration itself is bound by the depth of the deepest function in the AST, giving a critical path equal to the deepest function. In common cases where the depth of the AST is often fixed to a specific depth (source code often has a relatively small depth limit to the AST, despite very large code sizes), then the critical path might be considered constant in these cases. The worst-case scenarios for such algorithms are cases where there is a very deep tree with very small branching factors, leading to very little available parallelism.

This traversal idiom demonstrates a typical approach for working with the AST. Rather than leveraging structural recursion and a set of design patterns on top of that, a set of small idioms that represent specific actions apply over the AST and compose together. Developers may tweak the idioms to suit the needs of the specific compiler pass. The traversal idiom is the primary "walking" idiom. Other idioms focus on manipulation of the tree itself and identifying parts of the tree on which to do work.

An important "subterranean" technique that all these idioms leverage is the fact that node ids are the same values as the pointers to the nodes themselves. That is, a node id is a valid index into any of the columns of our AST. Additionally, the encoding permits arbitrary computation over node ids. The reification of node ids into an explicitly computable object, rather than an opaque pointer, is critical to the application and effective use of these idioms.

The final code for computing the nearest lexical contour `r` for each node in the AST is as follows:

```
r←I@{t[⍵]≠3}⍣≡⍨p
```

3.5. Lifting Functions

The `r` column created above represents a specialized traversal through the AST from any node to each of the enclosing function nodes that denote each lexical contour visible to the node. This relieves the burden of maintaining that structural information in the parent vector, so the compiler architecture can decouple lambda lifting from lexical resolution and simplify the structure of the AST much earlier. To simplify future passes, it is often beneficial to flatten the AST at the earliest opportunities. The Lift Functions pass flattens the tree so that function nodes appear as unique root nodes in the AST and not nested inside one another. Where a function sub-tree previously existed in the tree, a new variable node pointing to the lifted function root takes its place. This simplification of the tree reduces the depth of the tree to the maximum depth of any function. Since many following passes' critical paths depend on the depth of the AST, performing such flattening passes early produces a compounding benefit on future pass performance.

At its heart, function lifting demonstrates a simple lifting of subtrees and adding new nodes to replace the subtrees' original locations. This pass demonstrates lifting without the associated concerns of managing sibling relationships at the same time, since the lifted subtrees all move to the root level and have no ordering requirements. The previous pass demonstrated basic tree traversal but did not

modify the structure of the tree. Function lifting utilizes a simpler traversal but modifies the tree in significant ways.

Finally, namespaces in this treatment are treated just like functions of a different kind. This greatly simplifies the AST since the compiler does not need to explicitly handle a separate namespace node above and beyond the typical function nodes.

Function lifting adds a new variable node into the tree for each nested function (i.e., all functions that are not namespace functions) to replace the function sub-tree in its original location, serving as a reference to the newly lifted location. Adding nodes to a tree represented using the parent vector representation differs significantly from the traditional record representation. A record representation typically provides a node constructor for each type of node that independently allocates a new node and returns a reference/pointer to this newly created node. To link such new nodes into the tree, pointers serve to thread the nodes in the tree together, and many languages automatically handle the memory management for such objects through a garbage collector. In contrast, the parent vector representation maintains all nodes as a contiguously allocated region of memory (at least conceptually) and adding a new node into the tree is not obviously efficient: inserting a new element at the ends or in the middle of an array, thus changing the size of the array, could potentially result in copying the entire array to make space for the new element. This has obvious performance ramifications.

The nature of the function lifting pass permits the use of an idiomatic array catenation techniques to address this issue, because the parent vector need only preserve the relative ordering in

the vector of siblings and not of descendants, cousins, or ancestors. This partial ordering means that new elements/nodes may be added anywhere if they preserve sibling ordering. Lifting all functions to their own root level effectively eliminates any requirement to order siblings, since the semantics of the code treats all root nodes as independent in this regard. Both these properties combined allow new root-level function nodes to appear anywhere in the vector, giving the freedom to choose the most efficient means to extend the tree vectors with new elements/nodes.

Given a contiguously allocated vector, the most convenient means of inserting new elements is by adding them to the end. Well established methods exist to improve the amortized costs of this "tail catenation." In modern APL implementations, the idiomatic expression to extend an array `A` with elements `B` is `A,←B`. Importantly, APL implementations recognize this idiom and optimize it avoid the copying overheads of a naïve version. This enables efficient insertion of new nodes to the end of a parent vector efficiently without changing, abstracting, or complicating the array-based model of trees. If nodes must be added into the middle of a parent vector, other idioms must be used, but other sections will treat that subject explicitly while this section focuses on the simpler variant where tail-catenation suffices.

The only other significant consideration is updating field or edge data for elements already in the tree. In such cases, the code uses direct array mutation of the appropriate elements.

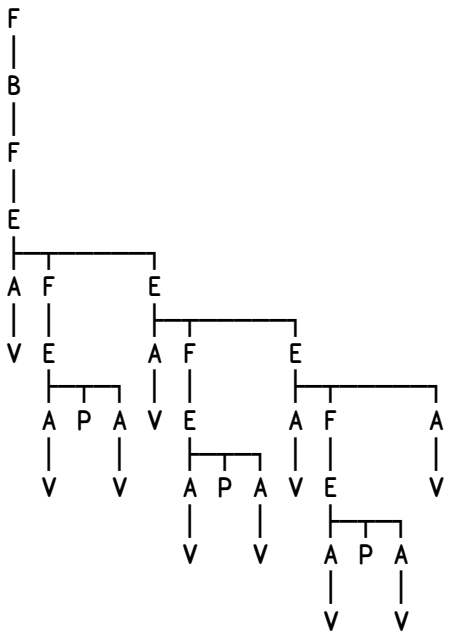
Thus, function lifting serves as a relatively simple example of the typical phases of a tree manipulation: 1) selecting nodes over which to operate; 2) adding new nodes to an AST; 3) modifying existing nodes, including type, edges, and other field data. In all these phases it is worth noting that

the fully raw and exposed encoding of trees as an integer parent vector enables the full and direct use of all the primitives of APL and the expressive power therein. When using APL primitives in this way, it may be useful to mentally map their names and definitions into the domain of trees, such that, for example, the primitive `⍸P` is read as “the nodes where property `P` holds” instead of reading “the non-zero indices of `P`.”

Consider the following source program:

```
:Namespace
  Fn←{⍺ {⍺+⍵} ⍺ {⍺+⍵} ⍺ {⍺+⍵} ⍵}
:EndNamespace
```

Parsing the above gives the following tree structure:

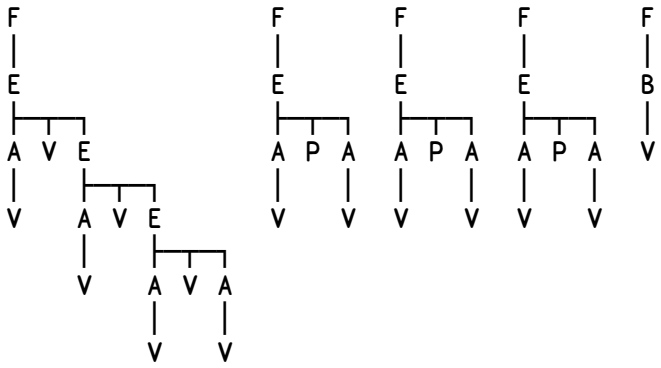


The parsed AST gives the following values for `d`, `p`, `t`, `k`, and `n`, indexed by node id `i`.

```
i 0  1 2 3 4  5 6 7 8  9 10 11 12 13 14 15 16 17 18 19
d 0  1 2 3 4  5 4 5 6  7  6  6  7  4  5  6  5  6  7  8
p 0  0 1 2 3  4 3 6 7  8  7  7 11  3 13 14 13 16 17 18
t 3  1 3 2 0 10 3 2 0 10  9  0 10  2  0 10  3  2  0 10
k 1  2 1 2 1  0 1 2 1  0  0  1  0  2  1  0  1  2  1  0
n 0 -5 0 0 0 -2 0 0 0 -2 -6  0 -1  0  0 -2  0  0  0 -2

       i 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34
       d  7  7  8  5  6  7  6  7  8  9  8  8  9  6  7
       p 17 17 21 13 23 24 23 26 27 28 27 27 31 23 33
       t  9  0 10  2  0 10  3  2  0 10  9  0 10  0 10
       k  0  1  0  2  1  0  1  2  1  0  0  1  0  1  0
       n -6  0 -1  0  0 -2  0  0  0 -2 -6  0 -1  0 -1
```

After lifting the function, the AST looks like this:




```
i 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21
p 0  0  1 35  3  4  3 36  7  8  7  7 11  3 13 14 13 37 17 18 17 17
t 3  1 10  2  0 10 10  2  0 10  9  0 10  2  0 10 10  2  0 10  9  0
k 1  2  1  2  1  0  1  2  1  0  0  1  0  2  1  0  1  2  1  0  0  1
n 0 -5 35  0  0 -2 36  0  0 -2 -6  0 -1  0  0 -2 37  0  0 -2 -6  0
r 0  0  0 35 35 35 35 36 36 36 36 36 36 35 35 35 35 37 37 37 37 37

       i 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38
       p 21 13 23 24 23 38 27 28 27 27 31 23 33 35 36 37 38
       t 10  2  0 10 10  2  0 10  9  0 10  0 10  3  3  3  3
       k  0  2  1  0  1  2  1  0  0  1  0  1  0  1  1  1  1
       n -1  0  0 -2 38  0  0 -2 -6  0 -1  0 -1  0  0  0  0
       r 37 35 35 35 35 38 38 38 38 38 38 35 35  0 35 35 35
```

Notice that after lifting, each function now resides in its own separate tree with the associated `F` node serving as the root node. Each tree is unconnected from the others in the main tree structure described by the `p` vector, but the `n` field connects the trees together through secondary references. Prior to

lifting, `n` contained only negative values corresponding to symbolic names appearing in the symbol table. During lifting, the lifted function is linked to its original location by replacing the original location of the function in the tree with a variable reference whose `n` field points to the function by node id instead of by name. The use of negative values for symbols and non-negative values for pointers into the parent vector, that is, for node ids, splits the `n` field into two mutually exclusive spaces where each node must occupy a point in only one of these spaces. This provides compression in the space requirements of the tree, but it also serves as a useful measure to simplify code and improve robustness at the same time, since it prevents a node from ever being both a name and id reference at the same time.

The first phase of function lifting identifies the nodes to lift, namely, all function (type 3) nodes not already at the top-level, named `i`. This selection process is simpler than traversing the parent vector as when computing `r` and requires only a test of function type and a test for being a non-root node:

```
i←⍳(t=3)∧p≠⍳≠p
```

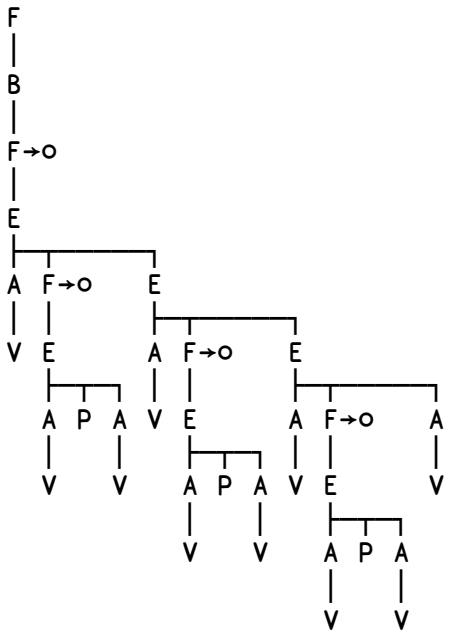
At this point, a subtle design consideration presents itself. This pass must manage two sets of nodes, the function nodes to lift and the variable nodes that take their place in the tree. The function nodes, as previously noted, do not depend on an ordering once lifted, but the variable nodes must appear in the same position in the AST as the original location of the corresponding function node. This means that the variable nodes do require a careful ordering. This presents two options when creating the new nodes. Typically, in traditional representations, the traversal accumulates function nodes into some

type of set to lift and insert higher up into the tree, while it generates new variable nodes during the traversal and links them to the parent of the corresponding function. In contrast, this pass generates a copy of each function node and catenates them to the end of the column vectors, such as the parent vector. It then modifies the original function node to a variable type and set its fields appropriately. Finally, the code links the function nodes' children to the new duplicate function node at the end of the tree columns. Putting the function nodes at the end of the parent vector and modifying the original locations to be variable nodes guarantees the ordering requirements for the variable nodes—which are identical to the original function node positions—without any additional computational costs, but also avoids any rearrangement of nodes, while adding the function nodes—which have essentially no ordering requirements—in the most efficient manner possible.

With the above strategy clarified and having already collected the ids of nodes that to lift, the following code adds the new nodes to the tree and, in preparation for them becoming variable nodes, links each original function node to its corresponding replacement node:

```
p,←n[i]←(≢p)+⍳≢i
```

The code says much more here than it may at first appear to say. This expression produces a tree associating each function node given in `i` with a newly allocated node linked to it via the `n` field as shown graphically in the following tree, where the newly allocated node is indicated as `o` because it has no type yet:



The expression divides into 3 major pieces: `p,←`, `n[i]←`, and `(≢p)+⍳≢i`. The two assignments both receive as their inputs the result of the 3rd piece `(≢p)+⍳≢i`. The piece `p,←` is a tail catenation to `p` and is the part that "allocates" the new nodes. The piece `n[i]←` assigns the links to the new nodes from the old function nodes `i`. Finally, the expression `(≢p)+⍳≢i` is the set of new node ids, that is, $id_{(\equiv p)}...id_{(\equiv p)+(\equiv i)-1}$ where `(≢p)` and `(≢i)` are the lengths of vectors `p` and `i`, respectively. There are some subtle features of the representation used to good effect here.

Note that one may interchange the positions of `p,←` and `n[i]←` without altering the meaning. The expression snippet `n[i]←p,←` works in the same manner as `p,←n[i]←`. This means that the links to the nodes can be created before any allocations for the nodes need take place. Furthermore, the expression `(≢p)+⍳≢i` gives the ids and pointers to the new nodes also before any allocation. In a traditional encoding, this is not possible because the pointers are usually meant to be treated opaquely and not computed directly. Working directly with the pointers enables the above bulk allocation. The

tail catenation requires only a single allocation. The above expression is idiomatic and easily optimized. The writes to `p` and `n` may both be written in bulk, parallel fashion. In the case of `p`, not only are the writes parallel, they are also contiguous, allowing for more performance. It is also worth noting that this expression works when there are no functions to lift and `i≡⍬`. The nodes' ids are the parent vector values because the newly lifted function nodes are root nodes and therefore their own parents.

The code also allocates and sets the `t`, `k`, `n`, and `r` fields of the new function nodes. Since these are inner function nodes, their type is 3, kind 1, with an empty `n` field and the same `r` field as the original function nodes. The code preserves the `r` field lest it lose the lexical contour information necessary for future passes. A simple way to allocate these fields might look like this:

```
t,←(≢i)⍴3
k,←(≢i)⍴1
n,←(≢i)⍴0
r,←r[i]
```

Instead of using four separate statements to accomplish these allocations, however, a single statement can allocate them as a group:

```
t k n r,←3 1 0(r[i])⍴⍨≢i
```

As for the nodes `i`, after the above transformations they still have a type and kind of 3 1, indicating function nodes, but they must change to variable nodes of type 10 1. One way to do this would be the following two expressions:

```
t[i]←10
k[i]←1
```

But a single expression can also work using the idiom `¬@i`, where `n¬@i⊢A` gives an array identical to `A` but with elements `A[i]` replaced with `n`:

```
t k(¬@i⍤)←10 1
```

There is now but one final need to address. At this point the variable nodes are in their correct position and new function nodes appearing at the root are in place and allocated, but none of the children of these functions point to these newly allocated function nodes as their parents, and the `r` field similarly points to the wrong nodes. At this point, nodes that should point to these new function nodes instead point to the original nodes, which are now variable nodes. The code must correct this before the tree will have the correct structure.

The code creates a redirect vector `V` such that `V[p]` and `V[r]` produce corrected `p` and `r` vectors. In the case where no modifications of the `p` or `r` vectors is required, then the index vector `⍳≠p` is a suitable redirect vector. If node `i` is now to be found at node `j`, then the element `V[i]` should be `j`. For all nodes `i`, the new nodes are given by `n[i]`, so starting with `⍳≠p` and replacing elements at indices `i` by elements `n[i]` gives a suitable redirect, for which the expression `n[i]@i⊢⍳≠p` suffices nicely. Thus, the code to correct the pointer vectors after moving the function nodes is the following:

```
p r I⍤←⊂n[i]@i⊢⍳≠p
```

This completes the pass. The entire complexity of this pass is linear in time and space but nearly all the operations are constant time critical path with the sole exception of the `⍸` (Where) in the node selection phase. Here is the pass in its entirety:

```
⍝ Lift Functions
p,←n[i]←(≢p)+⍳≢i←⍸(t=3)∧p≠⍳≢p ⋄ t k n r,←3 1 0(r[i])⍴⍨⍥≢i
p r I⍨←⊂n[i]@i⊢⍳≢p ⋄ t k (¬⍀i⍨)←10 1
```

3.6. Wrapping Expressions

A dfn consists of a series of statements, either guarded, bound, or unbound. A guarded statement consists of a test expression and a consequent expression. A bound expression is an expression whose final operation, in terms of execution order, is an assignment of some kind. An unbound expression is an expression that is neither a guarded nor a bound expression. Execution of a dfns consists of evaluating each statement in turn until a return condition is met. When a return condition is met, the value of the expression satisfying the return condition is returned as the value of the dfn's evaluation. The return conditions are any of the following that may occur during evaluation of the dfn:

1. A guard's test expression evaluates to 1, or true
2. An unbound expression is executed
3. The last expression is evaluated

This means that a dfn returns on the first true guard or the first unbound expression. When such a condition is met, no further statements are executed, meaning that functions such as `{5 ⋄ 3}` always evaluate to `5`, and the expression `3` never executes. The same holds for guards, so a guard such as in `{1:5 ⋄ 3}` that always tests true always returns `5` and the expression `3` never executes. Finally, in cases where neither a true guard nor an unbound expression executes before the last expression executes, the dfn returns the (shy) value of the last expression. If the last expression is a guard whose test is not true, the dfn returns no value.

In all cases, the dfns syntax has no explicit return syntax. All returns are implicit in the syntax. After function lifting, the Wrap Expressions pass introduces explicit return nodes for all statements indicating the nature of the statement, either a return or a continue statement. Return nodes are expressions of kind `0` or `¯1` that contain a single statement child. Wrapping statements pushes nodes down the tree and represents an opposite operation to lifting passes that move sub-trees up the tree instead of down.

Comparing the primary manipulation tasks between this and Function Lifting, the main elements of selecting nodes, adding nodes to the tree, and modifying edge data remain, but they involve more. The pass must select all potential return statements, which is not a simple selection on type and parent but also on sibling in the case of guard consequents. Moreover, and perhaps more critically, the pass inserts the return nodes into the middle of the tree in arbitrary position in order to wrap a statement.

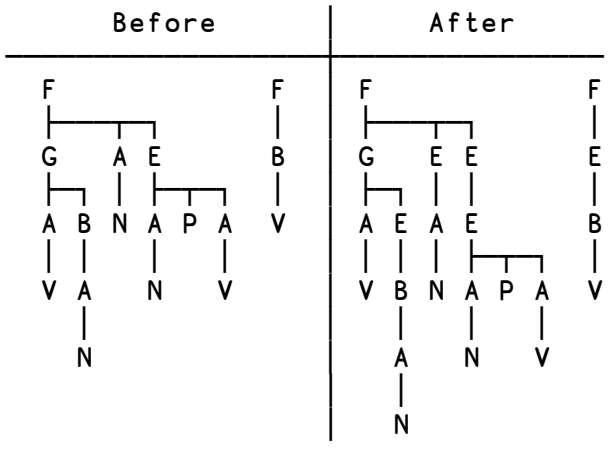
Function Lifting takes all function nodes to the top-level and therefore lacks sibling ordering requirements, allow the pass to catenate them to the end of the parent vector and other column vectors without issue. This pass could use the same techniques, but keeping in mind that a future pass needs to flatten all the complex nested expressions into simpler statements that preserve execution order, preserving the ordering in this pass simplifies that future pass by enforcing the same depth first pre-ordering within statement sub-trees. This requires a slightly more involved approach but improves future passes significantly.

Adding this ordering requirement means that the code must splice the new nodes into the middle of the column vectors. To do this efficiently, it must minimize the number of times it reorders nodes in any way that changes their pointers (node ids). In a very literal sense this requirement is equivalent to a requirement in a traditional approach to minimize garbage collections, and for much the same reasons. In this case, having explicit control over this process enables tight constraints on the number of reorderings.

The pass itself consists of four major phases: firstly, selecting the appropriate statement nodes; secondly, extending the vectors to create space for the new return nodes; thirdly, recomputing the pointer vectors after the reordering that must occur during extension; and finally, setting the appropriate type and kind of the newly created nodes and ensuring that the parent edges are appropriately handled. The following example illustrates the process:

```
:Namespace
  f←{⍵:x←3 ⋄ 5 ⋄ 2+⍵}
:EndNamespace
```

And here is what the AST looks like before and after the transformation:



Here are the column vectors before and after the transformation:

```
       before
i 0  1  2  3  4  5  6  7  8  9 10 11 12  13 14 15
p 0  0  1 15  3  3  5  6 15  8 15 10 11  10 10 15
t 3  1 10  4 10  1  0  7  0  7  2  0  7   9 10  3
k 0  1  1  0  0  0  0  0  0  0  2  0  0   1  0  1
n 0 -5 15  0 -1 -6  0 -7  0 -8  0  0 -9 -10 -1  0
r 0  0  0 15 15 15 15 15 15 15 15 15 15  15 15  0
       after
i 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16  17 18 19
p 0  0  1  2 19  4  4  6  7  8 19 10 11 19 13 14 15  14 14 19
t 3  2  1 10  4 10  2  1  0  7  2  0  7  2  2  0  7   9 10  3
k 0 -1  1  1  0  0  0  0  0  0  0  0  0  0  2  0  0   1  0  1
n 0 -5 -5 19  0 -1 -6 -6  0 -7  0  0 -8  0  0  0 -9 -10 -1  0
r 0  0  0  0 19 19 19 19 19 19 19 19 19 19 19 19 19  19 19  0
```

Note that the pass preserves the relative ordering of the various statement sub-trees, but at the cost of recomputing the parent vector and other pointer vectors. In Function Lifting, the vectors remained largely the same, with some additional data added to the end and some of the elements within changed. This pass recomputes the parent vector, referent, and name vectors.

The first phase of wrapping expressions selects the nodes to wrap. This includes all expressions that appear as immediate children of a function node and the consequent statements of any guard nodes. This includes the “namespace” function node(s) but does not include the test statement of the guard node. Thus, in the example, this corresponds to nodes 1, 5, 8, and 10, which are the binding nodes for variables `f` and `x`, as well as the two unbound expressions in the body of function `f`. The code breaks into two sub-tasks to select these nodes: selecting the function statements and selecting the consequent of any guard statements.

Selecting function statements uses the same basic techniques demonstrated in Function Lifting, as shown in the following progression:

```
      p
0 0 1 15 3 3 5 6 15 8 15 10 11 10 10 15
      t
3 1 10 4 10 1 0 7 0 7 2 0 7 9 10 3
      t[p]
3 3 1 3 4 4 1 0 3 0 3 2 0 2 2 3
      t[p]=3
1 1 0 1 0 0 0 0 1 0 1 0 0 0 0 1
      t∊3 4
1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1
      ~t∊3 4
0 1 1 0 1 1 1 1 1 1 1 1 1 1 1 0
      (~t∊3 4)∧t[p]=3
0 1 0 0 0 0 0 0 1 0 1 0 0 0 0 0
      ⍸(~t∊3 4)∧t[p]=3
1 8 10
```

Recall that types 3 and 4 correspond to the type of function nodes and guard nodes, respectively.

Selecting the guard consequents involves a different technique. At this point the guard nodes retain their relative ordering given to them at parse time. This means that every guard has, in order, a test statement, followed by a single consequent statement. Selecting every second node whose parent is a guard therefore amounts to selecting the consequent expressions, as shown in the following progression:

```
      t[p]=4
0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0
      ⍸t[p]=4
4 5
      ≠⍸t[p]=4
2
      ⍳≠⍸t[p]=4
0 1
      2|⍳≠⍸t[p]=4
0 1
      0 1≠⍸t[p]=4
5
      {⍵≠⍨2|⍳≠⍵}⍸t[p]=4
5
```

Thus, the following composition computes the set of nodes `i` that contains the set of all nodes that need to be wrapped in return nodes:

```
      ⊢i←(⍳⍨(~t∊3 4)∧t[p]=3),{⍵≠¨2|⍳≠⍵}⍳⍨t[p]=4
1 8 10 5
```

The second phase of wrapping expressions makes room for the return nodes to be inserted into the tree. Recall the need to preserve pre-order depth-first ordering that currently retained in each statement sub-tree. Each node in vector `i` needs precisely one return node. To preserve ordering, this node must appear directly atop, that is, just before, the statement node that it wraps. The code leverages the Replicate (⍿) function to achieve this. This function receives as its left argument a vector wherein each element indicates the number of times to repeat the corresponding element of the right argument within the result vector. When used with a Boolean left argument, the replicate function filters its right argument to return only elements corresponding to non-zero elements in the left argument. However, the code uses replicate with a left argument containing values of either 1 or 2. For nodes that are not in `i`, it replicates the node only a single time (using a value of 1 in the left argument), but for the nodes to be wrapped, it duplicates each node contiguously, so that two occurrences of the node appear adjacent in the result. This creates space in the right position for return nodes. However, it also pushes nodes down through the vector, giving them different node ids, and potentially invalidating the pointer vectors. Thus, phase three rectifies this once the code has expanded the columns.

The replication vector above is vector of 1's with the exception that elements `i` are 2's. The use of `@` serves well in this capacity:

```
      i
1 8 10 5
      1⍴⍨≠p
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
      2@i⊢1⍴⍨≠p
1 2 1 1 1 2 1 1 2 1 2 1 1 1 1 1
```

This replication vector works for all columns, but it also indicates the relative offsets between nodes and is needed in the 3rd phase to recompute the pointer vectors, so it is saved `m`. Applying `m` to replicate on each column leads to the following:

```
      m←2@i⊢1⍴⍨≠p
      m≠p
0 0 0 1 15 3 3 3 5 6 15 15 8 15 15 10 11 10 10 15
      m≠t
3 1 1 10 4 10 1 1 0 7 0 0 7 2 2 0 7 9 10 3
      m≠k
0 1 1 1 0 0 0 0 0 0 0 0 0 2 2 0 0 1 0 1
      m≠n
0 ¯5 ¯5 15 0 ¯1 ¯6 ¯6 0 ¯7 0 0 ¯8 0 0 0 ¯9 ¯10 ¯1 0
      m≠r
0 0 0 0 15 15 15 15 15 15 15 15 15 15 15 15 15 15 0
      p t k n r≠⍨←⊂m←2@i⊢1⍴⍨≠p
      ↑p t k n r
0  0  0  1 15  3  3  3  5  6 15 15  8 15 15 10 11  10 10 15
3  1  1 10  4 10  1  1  0  7  0  0  7  2  2  0  7   9 10  3
0  1  1  1  0  0  0  0  0  0  0  0  0  2  2  0  0   1  0  1
0 ¯5 ¯5 15  0 ¯1 ¯6 ¯6  0 ¯7  0  0 ¯8  0  0  0 ¯9 ¯10 ¯1  0
0  0  0  0 15 15 15 15 15 15 15 15 15 15 15 15 15  15 15  0
```

The 3rd phase recomputes the now invalid pointer vectors. These are the `p`, `r`, `i`, and `n` variables. It uses the same basic technique as in the previous section: it computes a redirect vector `j` such that `j[v]` for pointer vector `v` is the revalidated `v`. The replication vector `m` stores this information in relative form. One way to see `m` is as an offset vector indicating for each element its index in the post-

replication vector relative to the index of the element before it. Thus, for element `k`, its index is `(m[k]+j[k-1])-1`. This is just a description of a prefix sum, so the following readily computes `j`:

```
      (+⍀m)-1
0 2 3 4 5 7 8 9 11 12 14 15 16 17 18 19
```

The pointer vectors `p`, `r`, and `i` recompute directly as:

```
      j←(+⍀m)-1
      j[p]
0 0 0 2 19 4 4 4 7 8 19 19 11 19 19 14 15 14 14 19
      j[r]
0 0 0 0 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19 0
      j[i]
2 11 14 7
      p r i I⍤←⊂j←(+⍀m)-1
      ⍪p r i
```

0 0 0 2 19 4 4 4 7 8 19 19 11 19 19 14 15 14 14 19
0 0 0 0 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19 0
2 11 14 7

However, the `n` column does not admit a direct recomputation because it contains negative values, and only the positive elements should be recomputed. This can be done by using `@` like so:

```
      n
0 ¯5 ¯5 15 0 ¯1 ¯6 ¯6 0 ¯7 0 0 ¯8 0 0 0 ¯9 ¯10 ¯1 0
      j
0 2 3 4 5 7 8 9 11 12 14 15 16 17 18 19
      0≤n
1 0 0 1 1 0 0 0 1 0 1 1 0 1 1 1 0 0 0 1
      (0≤⊢)n
1 0 0 1 1 0 0 0 1 0 1 1 0 1 1 1 0 0 0 1
      j I@(0≤⊢)n
0 ¯5 ¯5 19 0 ¯1 ¯6 ¯6 0 ¯7 0 0 ¯8 0 0 0 ¯9 ¯10 ¯1 0
      n←j I@(0≤⊢)n
```

This results in valid pointer vectors at the end of phase 3.

The final 4th phase fixes up the nodes and edges yet to be fixed. Namely, nodes `i` must now point to the newly allocated `i-1` return nodes, and nodes `i-1` must have the correct type and kind. Pointing nodes `i` to nodes `i-1` wraps nodes `i` under nodes `i-1`, as follows:

```
      i
2 11 14 7
      i-1
1 10 13 6
      p
0 0 0 2 19 4 4 4 7 8 19 19 11 19 19 14 15 14 14 19
      p[i]←j←i-1
      p
0 0 1 2 19 4 4 6 7 8 19 10 11 19 13 14 15 14 14 19
```

Saving nodes `i-1` as `j` preserves them for upcoming expressions. The node kinds `k[j]` should be either `0` or `¯1` depending on whether the return node indicates a return or not, respectively. Nodes that are non-final binding nodes or nodes that are top-level are all non-return and indicated by a kind of `¯1`, while all others are returns indicated by `0`.

The referent kind helps to determine top levelness:

```
      j
1 10 13 6
      r
0 0 0 0 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19 0
      k
0 1 1 1 0 0 0 0 0 0 0 0 2 2 0 0 1 0 1
      r[j]
0 19 19 19
      k[r[j]]
0 1 1 1
      k[r[j]]=0
1 0 0 0
```

The expression `t[j]=1` gives nodes of binding type (type 1). To exclude the final, that is, rightmost nodes, in the case of final nodes that are also binding nodes, note that the last element of a group of

stably grouped sibling nodes is the final node of the group and it is the first node if reversed. This suggests the use of the group-by, or Key, operator (⌸) as follows:

```
      p[j]
0 19 19 4
      ⊢∘⊃1 2 3
1
      ⊢∘⊃5 3 7
5
      {⍺⍵}⌸p[j]
┌──┬───┐
│0 │0  │
├──┼───┤
│19│1 2│
├──┼───┤
│4 │3  │
└──┴───┘
      {⊃⌽⍵}⌸x
0 2 3
```

This gives the final nodes from nodes `j`. Using this to mask out the final nodes from `t[j]=1`:

```
      j
1 10 13 6
      t
3 1 1 10 4 10 1 1 0 7 0 0 7 2 2 0 7 9 10 3
      t[j]
1 0 2 1
      t[j]=1
1 0 0 1
      0@({⊃⌽⍵}⌸p[j])⊢t[j]=1
0 0 0 0
```

The new values for `k[j]` are just the negation of the logical OR-ing of the mask of top-level nodes and that of non-final binding nodes:

```
      k[r[j]]=0
1 0 0 0
      0@({⊃⌽⍵}⌸p[j])⊢t[j]=1
0 0 0 0
      -(k[r[j]]=0)∨0@({⊃⌽⍵}⌸p[j])⊢t[j]=1
¯1 0 0 0
      k[j]←-(k[r[j]]=0)∨0@({⊃⌽⍵}⌸p[j])⊢t[j]=1
```

And finally, return nodes all have expression type:

```
      t[j]←2
      t
3 2 1 10 4 10 2 1 0 7 2 0 7 2 2 0 7 9 10 3
```

The complexity of this pass is linear in both time and space, and nearly all operations are constant time critical paths. The ⍸, ⍀, and ⌿ operations are logarithmic critical path. In the selection phase this logarithmic path is in the size of the tree, but in all other statements the input size is the number of statements appearing in functions, including guard statements.

Here is the complete pass:

```
⍝ Wrap Expressions
i←(⍸(~t∊3 4)∧t[p]=3),{ω⌿⍨2|⍳≢ω}⍸t[p]=4 ⋄ p t k n r⌿⍨←⊂m←2@i⊢1⍴⍨≢p
p r i I⍨←⊂j←(+⍀m)-1 ⋄ n←j I@(0≤⊢)n ⋄ p[i]←j←i-1
k[j]←-(k[r[j]]=0)∨0@({⊃⌽ω}⌸p[j])⊢t[j]=1 ⋄ t[j]←2
```

3.7. Lifting Guard Test Expressions

The parser represents one-armed conditional statements using Guard nodes (type 4). After parsing, each guard has two children and no `n` field data. The first (leftmost) child is the test expression and the second (rightmost) is the consequent expression to execute if the test expression returns true (1).

The Lift Guard Tests pass simplifies the structure of the guard nodes by lifting test expressions out of guards and making them immediate left siblings of the guard nodes. This prepares the way to lift and flatten nested expressions in a future pass. The pass consists of the following conceptual steps;

1. Selecting test expressions and their guards

2. Updating parents

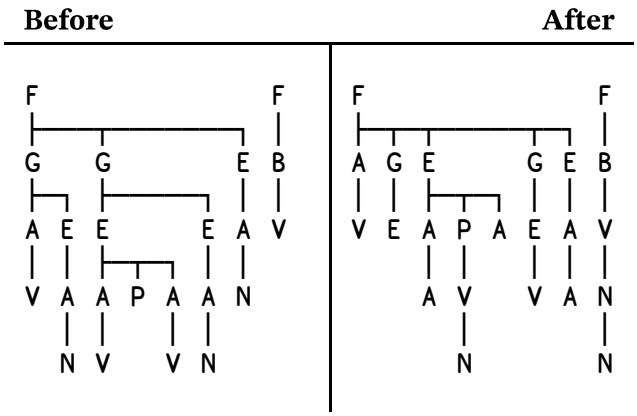
3. Positioning siblings

4. Recomputing the parent vector

As an example, consider the following program containing two guard statements:

```
:Namespace
  f←{⍵:0 ⋄ ⍺+⍵:1 ⋄ 2}
:EndNamespace
```

With its AST before and after lifting guard tests:



And the tree columns before and after lifting:

```
       before
i 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
p 0  0  1  2 20  4  4  6  7 20  9 10 10 10  9 14 15 20 17 18 20
t 3  2  1 10  4 10  2  0  7  4  2 10  9 10  2  0  7  2  0  7  3
k 0 ¯1  1  1  0  0  0  0  0  0  2  0  1  0  0  0  0  0  0  0  1
n 0 ¯5 ¯5 20  0 ¯1  0  0  0  0  0 ¯2 ¯6 ¯1  0  0 ¯7  0  0 ¯8  0
r 0  0  0  0 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20  0
       after
i 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
p 0  0  1  2 20 20  5  6  7 20 20  9  9  9 10 14 15 20 17 18 20
t 3  2  1 10 10  4  2  0  7  2  4 10  9 10  2  0  7  2  0  7  3
k 0 ¯1  1  1  0  0  0  0  0  2  0  0  1  0  0  0  0  0  0  0  1
n 0 ¯5 ¯5 20 ¯1 ¯1  0  0  0  0  0 ¯2 ¯6 ¯1  0  0 ¯7  0  0 ¯8  0
r 0  0  0  0 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20  0
```

The test nodes are selected using the same technique as in the previous Wrap Return Expressions pass selecting the leftmost node instead of the rightmost. This node set is bound to `i`. The corresponding guard nodes are just the parents of `i`, which called `x`:

```
      p
0 0 1 2 20 4 4 6 7 20 9 10 10 10 9 14 15 20 17 18 20
      t
3 2 1 10 4 10 2 0 7 4 2 10 9 10 2 0 7 2 0 7 3
      t[p]
3 3 2 1 3 4 4 2 0 3 4 2 2 2 4 2 0 3 2 0 3
      t[p]=4
0 0 0 0 0 1 1 0 0 0 1 0 0 0 1 0 0 0 0 0 0
      ⍸t[p]=4
5 6 10 14
      ~2|⍳≠⍸t[p]=4
1 0 1 0
      {⍵≠⍤¯1⊢~2|⍳≠⍵}⌸t[p]=4
5 10
      ⊢x←p[i←{⍵≠⍤¯1⊢~2|⍳≠⍵}⌸t[p]=4]
4 9
```

Next, nodes `i` must have new parents, specifically, the same parents as their containing guards:

```
p[i]←p[x]
```

At this point, the relative order in the columns means that the test expression is now the immediate right sibling of its associated guard, instead of the intended left sibling. Swapping the guard and test nodes ensures the correct ordering, conveniently without need to manipulate of any other nodes. Note that `p[i]≡p[x]` and that `r[i]≡r[x]`, so these columns require no modification. Moreover, because the n field of a guard node contains no meaningful data, only the `n` field of the test expressions needs to be swapped. The following statements swap other guard and test fields in the tree:

```
t[i,x]←t[x,i] ⋄ k[i,x]←k[x,i] ⋄ n[x]←n[i]
```

With test and guard nodes where they should be, the parent vector must point the children of these nodes to the new locations. This recomputation uses the same technique as in previous passes:

```
p←((x,i)@(i,x)⊢⍳≠p)[p]
```

The complexity of this pass is trivially linear in time and space with constant time critical path in all cases but in selecting the test nodes, which is logarithmic critical path in the size of the tree.

The complete pass is as follows:

```
⍝ Lift Guard Test Expressions
p[i]←p[x←¯1+i←{ω≠⍨⍵~2|⍳≠⍵}⍸t[p]=4] ⋄ t[i,x]←t[x,i] ⋄ k[i,x]←k[x,i]
n[x]←n[i] ⋄ p←((x,i)@(i,x)⊢⍳≠p)[p]
```

3.8. Counting Rank of Index Operations

Most expressions in the AST fix the number of their children. However, indexing expressions may have any number of children. When a future pass flattens all the expressions, the parent vector will no longer provide enough information to determine which expressions appeared within an indexing expression. To preserve this information, the Count Index Expression Rank pass records the number of children an index expression has in its `n` field. This is a pure analysis pass that demonstrates a simple analysis over a node's children. While simple, it clarifies a sometimes non-obvious task of working on the children of a node when the parent vector appears to only facilitate walking up the tree as opposed to walking down it. The complete pass is as follows:

```
n[p≠⍨(t[p]=2)∧k[p]=3]+←1
```

It takes advantage of the semantics of APL's modified assignment operator. When evaluating an assignment like `n[i]+←1`, if `i` contains duplicate indices, then the corresponding element in `n` will increment multiple times in a stable manner; there are no race conditions permitted in the statement's execution. This has the effect of computing a histogram or count of the elements in `i` and storing them

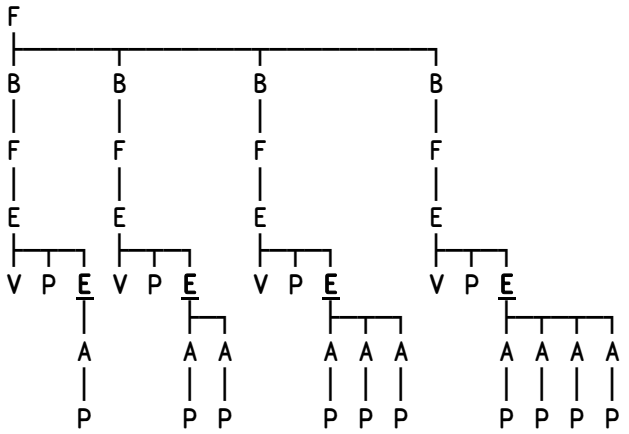
in a histogram vector. Alternatively, to storing the histogram data in `n`, the function `,∘≠⌸` applied monadically to `i` returns a histogram table instead of storing it into a vector.

```
,∘≠⌸p⌿⍨(t[p]=2)∧k[p]=3
```

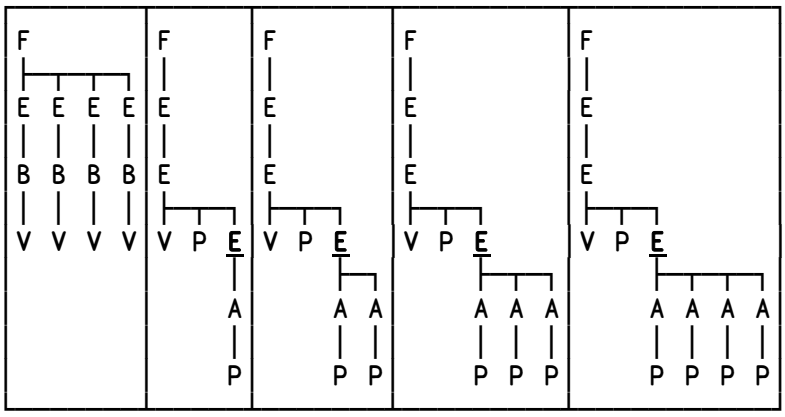
The following example demonstrates the results:

```
:Namespace
  R1←{⍵[]}
  R2←{⍵[;]}
  R3←{⍵[;;]}
  R4←{⍵[;;;]}
:EndNamespace
```

This example has four expressions that each have a different number of indexing places, separated by the semi-colons. When rendered in tree form, the four indexing nodes become apparent, along with their indexing spans. These are the nodes that will be annotated with the count of their children.



Right before the CI pass executes, the tree and data look like this, with the indexing nodes highlighted as bold in the column data:




```
i 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19
p 0  0  1  2 53  4  5  5  5  8  9  0 11 12 54 14 15 15 15 18
t 3  2  1 10  2  2 10  9  2  0  9  2  1 10  2  2 10  9  2  0
k 0 -1  1  1  0  2  0  1  3  3  0 -1  1  1  0  2  0  1  3  3
n 0 -5 -5 53  0  0 -1 -6  0  0 -7 -8 -8 54  0  0 -1 -6  0  0
r 0  0  0  0 53 53 53 53 53 53 53  0  0  0 54 54 54 54 54 54
```

```
   i 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36  37  38 39
   p 19 18 21  0 23 24 55 26 27 27 27 30 31 30 33 30 35   0  37 38
   t  9  0  9  2  1 10  2  2 10  9  2  0  9  0  9  0  9   2   1 10
   k  0  3  0 -1  1  1  0  2  0  1  3  3  0  3  0  3  0  -1   1  1
   n -7  0 -7 -9 -9 55  0  0 -1 -6  0  0 -7  0 -7  0 -7 -10 -10 56
   r 54 54 54  0  0  0 55 55 55 55 55 55 55 55 55 55 55   0   0  0
```

```
   i 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56
   p 56 40 41 41 41 44 45 44 47 44 49 44 51 53 54 55 56
   t  2  2 10  9  2  0  9  0  9  0  9  0  9  3  3  3  3
   k  0  2  0  1  3  3  0  3  0  3  0  3  0  1  1  1  1
   n  0  0 -1 -6  0  0 -7  0 -7  0 -7  0 -7  0  0  0  0
   r 56 56 56 56 56 56 56 56 56 56 56 56 56  0  0  0  0
```

And here is a breakdown of the pass operations:

```
       p
0 0 1  2 53 4 5 5 5 8 9 0 11 12 54 14 15 15 15 18 19 18 21 0
       23 24 55 26 27 27 27 30 31 30 33 30 35 0 37 38 56 40 41
       41 41 44 45 44 47 44 49 44 51 53 54 55 56
       t
3 2 1  10 2 2 10 9 2 0 9 2 1 10 2 2 10 9 2 0 9 0 9 2 1 10 2 2
       10 9 2 0 9 0 9 0 9 2 1 10 2 2 10 9 2 0 9 0 9 0 9 0 9 3
       3 3 3
       t[p]=2
0 0 1  0 0 1 1 1 1 1 0 0 1 0 0 1 1 1 1 1 0 1 0 0 1 0 0 1 1 1 1
       1 0 1 0 1 0 0 1 0 0 1 1 1 1 1 0 1 0 1 0 1 0 0 0 0 0
```

```
      k
0 ¯1 1 1 0 2 0 1 3 3 0 ¯1 1 1 0 2 0 1 3 3 0 3 0 ¯1 1 1 0 2 0
      1 3 3 0 3 0 3 0 ¯1 1 1 0 2 0 1 3 3 0 3 0 3 0 3 0 1 1 1
      1
      k[p]=3
0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 1 1 1 1 0 0 0 0 0 0 0 0
      1 1 1 1 1 1 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 0 0 0 0
      (t[p]=2)∧k[p]=3
0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0
      1 0 1 0 1 0 0 0 0 0 0 0 0 0 1 0 1 0 1 0 1 0 0 0 0 0
      ⍸(t[p]=2)∧k[p]=3
9 19 21 31 33 35 45 47 49 51
      p≠⍨(t[p]=2)∧k[p]=3]
8 18 18 30 30 30 44 44 44 44
      n[8 18 30 44]
0 0 0 0
      n[p≠⍨(t[p]=2)∧k[p]=3]+←1
      n[8 18 30 44]
1 2 3 4
```

The computational complexity is trivially linear in time and space. The critical path is bounded by the filter and histogram operation, which are both logarithmic in the size of the tree and number of indexing node children, respectively.

3.9. Flattening Expressions

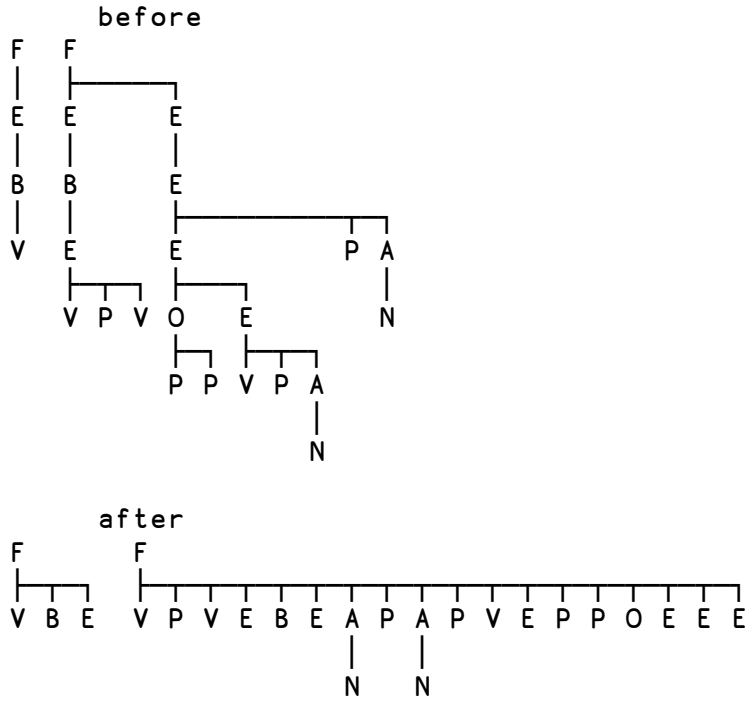
While not the most complicated or largest pass in the compiler, the process of flattening nested expressions is perhaps the most subtle in its simplicity. The goal of expression flattening is much the same as function lifting or guard lifting discussed in previous sections. The same phases of selection, permutations, and correction apply, but the permutation phase requires a seemingly involved manipulation. Previously, Function Lifting imposed no order requirement on the lifted nodes and used tail catenation to insert new nodes into the tree freely. Lifting guards only requires swapping/flipping two nodes relative to each other. Both these cases exhibit minimal order dependency. Nested expressions imply a specific order of execution as do any set of statements in a

function, and they must all be properly ordered relative to each other. In other words, Expression Flattening requires enforcing a total ordering on the lifted nodes within a function.

After lifting all expressions to the top level of their containing function node, all expression, atomic, and binding nodes appear in linear execution order, rather than relying on nesting to encode execution order. Unlike function lifting, where variable nodes that refer to the lifted nodes replace lifted nodes in their original positions, lifted expressions assume a stack semantics, and so require no new variable nodes as with function lifting. The precise requirements of expression ordering in APL will be discussed later, but for now, consider the following example of the entire pass:

```
:Namespace
  dist←{xy←⍺-⍵ ⋄ (+⌿xy*2)*.5}
:EndNamespace
```

With the following before and after trees:



And here are the before and after data:

```
       before
i 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17  18 19
p 0  0  1  2 24  4  5  6  6  6 24 10 11 12 13 13 12 16  16 16
t 3  2  1 10  2  1  2 10  9 10  2  2  2  8  9  9  2 10   9  0
k 0 ¯1  1  1 ¯1  0  2  0  1  0  0  2  1  2  2  1  2  0   1  0
n 0 ¯5 ¯5 24 ¯6 ¯6  0 ¯2 ¯7 ¯1  0  0  0  0 ¯8 ¯9  0 ¯6 ¯10  0
r 0  0  0  0 24 24 24 24 24 24 24 24 24 24 24 24 24 24  24 24

        20  21 22  23 24
        19  11 11  22 24
         7   9  0   7  3
         0   1  0   0  1
       ¯11 ¯10  0 ¯12  0
        24  24 24  24  0

       after
i 0  1  2  3  4  5  6  7  8  9 10  11 12  13 14 15 16 17 18
p 0  0  0  0 24 24 24 24 24 24 24  24 24  24 24 24 24 24 24
t 3 10  1  2 10  9 10  2  1  2  0   9  0   9 10  2  9  9  8
k 0  1  1 ¯1  0  1  0  2  0 ¯1  0   1  0   1  0  2  1  2  2
n 0 24 ¯5 ¯5 ¯1 ¯7 ¯2  0 ¯6 ¯6  0 ¯10  0 ¯10 ¯6  0 ¯9 ¯8  0
r 0  0  0  0 24 24 24 24 24 24 24  24 24  24 24 24 24 24 24

       19  20 21 22  23 24
       24  12 24 24  10 24
        2   7  2  2   7  3
        1   0  2  0   0  1
        0 ¯11  0  0 ¯12  0
       24  24 24 24  24  0
```

The selection phase comes first. Note that in APL, expressions are said to execute “right to left” while statements execute left to right. Thus, each statement must be lifted as a group and nodes from one statement must not appear mixed into or prior in the code to the nodes of another statement. Each statement serves as a sort of partitioning for the nodes it contains, so when selecting nodes to lift, the code must identify the root-level statement node that encloses the expressions. This is the node whose parent is either a guard or function node. Because of the total ordering requirement, the code must also select nodes already at the top level of a function. The node types to lift are guards, atomic,

binding, expression, operator, primitive, and variable nodes. This gives the following set of nodes `i` to lift:

```
      4,(⍳3),8+⍳3
4 0 1 2 8 9 10 ⍝ Guard, Atomic, Bind, Expr, Oper, Prim, Var
      t
3 2 1 10 2 1 2 10 9 10 2 2 2 8 9 9 2 10 9 0 7 9 0 7 3
      t∊4,(⍳3),8+⍳3
0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 0 0
      ⍸t∊4,(⍳3),8+⍳3
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 21 22
      i←⍸t∊4,(⍳3),8+⍳3
```

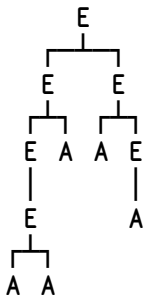
A variation of the traversal idiom used to compute `r` previously computes the root statement node for each node to lift, which called `x`:

```
      p[i]
0 1 2 24 4 5 6 6 6 24 10 11 12 13 13 12 16 16 16 11 11
      t[p[i]]
3 2 1 3 2 1 2 2 2 3 2 2 2 8 8 2 2 2 2 2 2
      t[p[i]]∊3 4
1 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0
      ~t[p[i]]∊3 4
0 1 1 0 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1
      p I@{~t[p[i]]∊3 4}⍤≡i
1 0 0 4 24 24 24 24 24 10 24 24 24 24 24 24 24 24 24 24 24
      x←p I@{~t[p[i]]∊3 4}⍤≡i
```

The next phase is manipulation. The simple part is to lift all the nodes to the function root level, which is trivially accomplished using `x` as follows:

```
p[i]←p[x]
```

After the appropriate lifting, the order of the nodes is not correct, and herein lies the subtlety of this pass. The ordering requirements of APL are that statements within a function execute in order from left to right and that each statement executes right to left. To see this impact on node ordering, consider the following example statement:



At this point, after updating the `p` column, these nodes remain in their depth-first pre-order traversal ordering:

```
E E E E A A A E A E A
```

But the ordering according to the execution order of the tree is thus:

```
A E A E A A A E E E E
```

Notice that this is precisely the reverse of the previous ordering. The primary ordering requirement for the lifted expressions reduces to reversing the nodes relative to one another within a statement while preserving the original order between statements.

The general strategy leverages a permutation vector `j` that reorders nodes `i`. To do this, first note that the nodes of `x`, that is, the statements, are properly ordered when arranged in ascending order by node id. Second, note that the ascending sort permutation given by ⍋ is a stable sort. This means that by reversing the nodes `i` and then sorting them in ascending order by statement id, the order of the nodes will match the desired execution order. This gives the following expression for `j`:

```
      x
1 0 0 4 24 24 24 24 24 10 24 24 24 24 24 24 24 24 24 24 24
      ϕx
24 24 24 24 24 24 24 24 24 24 24 10 24 24 24 24 24 4 0 0 1
      ⍋ϕx
18 19 20 17 11 0 1 2 3 4 5 6 7 8 9 10 12 13 14 15 16
```

```
       i
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 21 22
       ϕi
22 21 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
       (ϕi)[⍋ϕx]
3 2 1 4 10 22 21 19 18 17 16 15 14 13 12 11 9 8 7 6 5
       j←(ϕi)[⍋ϕx]
```

Using this to reorder the column vectors:

```
p[i]←p[j] ⋄ t[i]←t[j] ⋄ k[i]←k[j] ⋄ n[i]←n[j] ⋄ r[i]←r[j]
```

This completes the manipulation phase, but the parent vector still requires fixing up to maintain valid pointers. In this case, every node in `j` was moved to position `i`, so recomputation is as follows:

```
p←(i@j⊢⍳≠p)[p]
```

In complexity, most of the operations in this pass are linear time and space with constant critical path. The single sort is logarithmic critical path in the number of expression nodes. The traversal pattern during the selection phase is limited by the maximum depth of the functions, but when dealing with typical source code this is bound in practice.

The complete pass is as follows:

```
p[i]←p[x←p I@{~t[p[⍵]]∊3 4}⍤≡i←⍸t∊4,(⍳3),8+⍳3] ⋄ j←(ϕi)[⍋ϕx]
p t k n r{α[⍵]@i⊢α}←⊂j ⋄ p←(i@j⊢⍳≠p)[p]
```

3.10. Associating Frame Slots and Variables

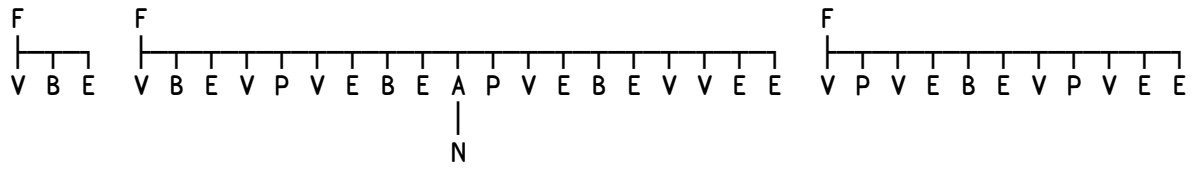
With expressions and functions flattened and simplified, the major remaining task is to resolve the free and local variable names. To facilitate this, the compiler computes some auxiliary information, primarily in the form of an `f` and `s` vector. This section covers the `s` vector and the next section covers the `f` vector.

Both the `f` and `s` vectors are defined relative to an `e` table that contains a listing of variable bindings and the functions in which they are bound. The `s` vector maps these bindings to slot indices. Thus, each function receives a frame containing slots addressed by index, and all bindings map to these slots.

Here is an example program:

```
:Namespace
  f←{g←{a←x+⍵ ⋄ a+a} ⋄ x←⍺+⍵ ⋄ y←x×2 ⋄ g y}
:EndNamespace
```

At this point in the compiler the tree looks like this:



And the desired values for `e` and `s`:

```
      e           s
 0  ¯5     0 0 1 2 0 ¯1
35 ¯10
35  ¯8
35  ¯6
36  ¯7
```

To begin, first compute `rn`, which is a table of bindings `b` and the function in which it appears:

```
       t
3 10 1 2 10 1 2 10 9 10 2 1 2 10 9 10 2 2 10 9 10 2 1 2 0 9
       10 2 1 2 7 10 10 2 2 3 3
       t=1
0 0 1 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1 0 0
       0 0 0 0 0 0
       ⍸t=1
2 5 11 22 28
       b←⍸t=1
       n[b]
¯5 ¯6 ¯7 ¯8 ¯10
```

```
      r[b]
0 35 36 35 35
      r[b],⍪n[b]
 0  ¯5
35  ¯6
36  ¯7
35  ¯8
35 ¯10
      rn←r[b],⍪n[b]
```

The `e` table is simply the unique rows of `rn` in sorted order:

```
      ⍋rn
0 4 3 1 2
      I∘⍋⍨rn
 0  ¯5
35 ¯10
35  ¯8
35  ¯6
36  ¯7
      ∪I∘⍋⍨rn
 0  ¯5
35 ¯10
35  ¯8
35  ¯6
36  ¯7
      e←∪I∘⍋⍨rn
```

Next, annotate the function nodes with the size of their frames and generate `s` based on a histogram of the first column (the `r` column) of `e`:

```
      0⌷⍉e
0 35 35 35 36
      x←0⌷⍉e
      ⊢∘≠⌸x
1 3 1
      ⍳⍨⊢∘≠⌸x
 0  0 1 2  0
      ∈⍳⍨⊢∘≠⌸x
0 0 1 2 0
      ¯1,⍨∈⍳⍨⊢∘≠⌸x
0 0 1 2 0 ¯1
      ∪x
0 35 36
      s←¯1,⍨∈⍳⍨n[∪x]←⊢∘≠⌸x
```

Notice the `¯1` catenated to the tail of `s`. This represents the "default" slot: when searching for an appropriate slot fails the result should be `¯1`. The use of this can be seen in the Anchor Variables (Lexical Resolution) pass described in a following section.

This pass is linear in space and time but contains `⍳⍨t=1` which is logarithmic time critical path in the size of the tree. Computing `e` uses a unique over a sort, so it is logarithmic time critical path in the size of `rn`, which is the number of bindings in the tree. The computation of `s` uses the `∊⍳⍨` idiom as well as `∪` and `⊢∘≠⌸`, all of which are logarithmic critical path in the number of unique bindings in the tree.

Here is the full pass:

```
s←¯1,⍨∊⍳⍨n[∪x]←⊢∘≠⌸x←0⌈⍀e←∪I∘⍋⍨rn←r[b],⍪n[b←⍳⍨t=1]
```

3.11. Placing Frames into a Lexical Stack

In addition to mapping unique bindings to frame slots, the code must determine the relative lexical position for each function. This is necessary to ensure that the variable reference to a binding can find its appropriate match within the lexical environments on the call stack. The `f` vector stores this mapping.

Taking the same example from last section, the `f` vector would look like this:

```
      f
0 1 1 1 2 ¯1
```

The code computes the `f` vector from the `r` column. To do this, use `d` as a temporary cache to compute function depth. This is just the walking distance from a given function to its root containing function

along `r`. Since functions correspond one to one to lexical contours in the dfns syntax, this makes function depth equivalent to frame or lexical depth.

To begin with, we resize and zero `d`:

```
      d
0 1 2 3 4 5 6 7 7 7 5 6 6 6 3 4 5 5 5 3 4 5 5 5 6 3 4 4 0 0 0 0 0
0 0 0 0
      t
3 10 1 2 10 1 2 10 9 10 2 1 2 10 9 10 2 2 10 9 10 2 1 2 0 9 10 2 1
2 7 10 10 2 2 3 3
      t=3
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 0
0 0 1 1
      ⍳t=3
0 35 36
      i←⍳t=3
      d[i]←0
```

Next, walk up the `r` vector for each function `i` and increment `d[i]` for each non-root node:

```
      r[i]
0 0 35
      i≠r[i]
0 1 1
      _←{z⊣d[i]+←⍵≠z←r[⍵]}⍣≡i
      d
0 1 2 3 4 5 6 7 7 7 5 6 6 6 3 4 5 5 5 3 4 5 5 5 6 3 4 4 0 0 0
      0 0 0 0 1 2
```

Finally, extract the appropriate depth for each unique binding in `e`. Using `¯1` as a default value:

```
      0⌷⍉e
0 35 35 35 36
      d[0⌷⍉e]
0 1 1 1 2
      d[0⌷⍉e],¯1
0 1 1 1 2 ¯1
      f←d[0⌷⍉e],¯1
```

The complexity of this pass is linear in space and time, and the critical path is dominated by the `⍳` and the longest function chain in `r`. Thus, this pass is logarithmic critical path in the size of the tree.

Here is the complete pass:

```
d←(≠p)↑d ⋄ d[i←⍸t=3]←0 ⋄ _←{z⊣d[i]+←ω≠z←r[ω]}⍣≡i ⋄ f←d[0]⌽e],¯1
```

3.12. Recording Exported Names

As a final preparatory pass before lexical resolution, the compiler records the names of bindings that appear at the top level of the source. This information would be lost during lexical resolution, so this pass preserves it before resolution occurs. This is the simplest pass in the entire compiler and demonstrates how much simpler constructs may be used in cases where the information or traversal requirements are simple, as they often are.

This pass requires just the `n` field of the binding nodes (type 1) whose nearest containing function is a namespace function (kind 0). Using the same example as the previous two sections, which has a single top-level binding:

```
       t
3 10 1 2 10 1 2 10 9 10 2 1 2 10 9 10 2 2 10 9 10 2 1 2 0 9
       10 2 1 2 7 10 10 2 2 3 3
       t=1
0 0 1 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1 0 0
       0 0 0 0 0 0
       r
0 0 0 0 35 35 35 36 36 36 36 36 36 36 36 36 36 36 35 35 35 35
       35 35 35 35 35 35 35 35 35 35 35 35 35 0 35
       k
0 1 1 ¯1 1 1 ¯1 0 1 0 2 0 ¯1 0 1 0 2 0 0 1 0 2 0 ¯1 0 1 0 2 0
       ¯1 0 0 1 1 0 1 1
       k[r]
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 1 1 1 1
       1 1 1 1 0 1
       k[r]=0
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 0
       0 0 0 0 1 0
       (t=1)∧k[r]=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 0 0
```

```
      n≠⍤⍥(t=1)∧k[r]=0
¯5
      xn←n≠⍤⍥(t=1)∧k[r]=0
```

This pass is trivially linear in time and space as well as logarithmic critical path.

3.13. Lexical Resolution

The dfns syntax is lexically scoped and obeys a stack discipline for binding visibility. Each function definition using brace notation introduces a new lexical scope/contour. Bindings introduced within a dfn are visible to all subsequent dfn bodies within it, but the binding does not exit the enclosing dfn. Arguments to functions are passed by value and not by reference in all cases. Consider the following examples:

```
      {X←5 ⋄ X}⍬
5
      {X←5 ⋄ Y←X+X ⋄ X←6 ⋄ Y+X}⍬
16
      {X←5 ⋄ f←2∘×⍤⍥X ⋄ X←4 ⋄ f X}⍬
128
      {X←5 ⋄ f←{α{α+X+ω}ω} ⋄ g←X∘f ⋄ X←4 ⋄ g 3}⍬
12
      {X←5 ⋄ f←{Y←X+X ⋄ X←3 ⋄ Y×X} ⋄ f⍬}⍬
30
      {X←5 ⋄ o←{αα+Y+ωω ω} ⋄ f←X o {Y←3 ⋄ X o{5+ω}ω} ⋄ f 7⊣X←Y←1}⍬
20
```

Here are some things to note specifically from the above examples:

1. Functions may include forward references to bindings.
2. Expressions may refer to bindings both to the left and to the right in the source text.
3. Names may be bound multiple times.

4. Operators resolve their operands at the position of the operator's call site, but free references inside of an operand resolve at the call sites of the derived functions, as are free references inside of the operators.
5. Free references resolve at the call sites of functions, not their definition sites.
6. An operator's operands may resolve to values inside of scopes that are below the visibility of an operator's free references. This implies that the semantics of operands are different than the semantics of free references.
7. Free references may have different values for every call to a given function.

The dfns syntax has a single binding form `x←e` that binds the name `x` to the value of executing `e`. In the full syntax, strand assignment permits more sophisticated structural binding that mirrors pattern matching in a small way, but to maintain simplicity this pass deals only with the simple name binding form. This binding form is also the name mutation form. Thus, if `x←e` appears in a function body after a prior binding to `x`, it does not introduce a new binding to `x` in a new lexical scope, but instead, it updates the existing binding to `x` in the same scope to point to `e` instead of its prior value.

Bindings are removed and deleted upon the return of their enclosing dfn. This works even in the presence of free references to a binding because a dfn may not return a function value, and thus, no binding will have a reference to it when its enclosing dfn returns, since any function that may have captured a reference to it will also be deleted by this time as well.

Therefore, the binding and name visibility rules for dfns are like Python without first class procedures, but with both higher order functions and free references.

The dfns binding rules also differ in another way from some Python implementations. In some Python implementations, all references to a name in each scope must refer to the nearest scope that contains a binding to that name. Thus, if a function binds `x` somewhere in its body, it will be an error to refer to `x` before the first binding to `x` appears in the body of the function, even if another binding to `x` appears in an enclosing scope. A binding to `x` in the body of a function shadows `x` for all the body of that function, and not just for statements appearing after it in the body.

In contrast to the above, within the dfns syntax, a reference to `x` before the first binding to `x` within the body of a function is considered a free reference. This admits the following program:

```
      {x←5 ⋄ {x←x+x ⋄ x×x}⍬}⍬
100
```

Finally, the value of a free reference to a binding for a given dfn is the value of the binding at the time the function is applied, as illustrated by the following program:

```
      {x←5 ⋄ f←{x} ⋄ x←3 ⋄ f⍬}⍬
3
```

This is a consequence of the `x←3` statement affecting the same lexical scope as `x←5` and thus altering the binding to `x` instead of introducing a new lexical scope.

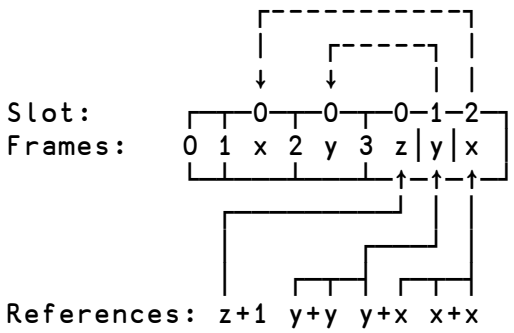
The purpose of name resolution is to associate each reference with a specific frame, which indicates which enclosing scope contains the binding to which the variable refers, and a specific slot, which indicates a specific numeric offset into the frame that stores the value. Each unique name bound within a dfn maps to a numeric slot in some frame. The nested structure of dfns forms the set of frames. The 0th frame is always the top-level, with the 1st frame being the bindings of one of the top-level

function definitions, and so on down the nested dfns definitions. Because dfns are not first-class procedures, there is no need to create closures; resolution need only ensure each function has access to its own stack frame and all its enclosing frames.

Resolution must handle the case described above where a reference to a name appears before the first binding to the same name within the same dfn body. To address this issue, treat each function body as containing an initialization statement at the beginning of the function for each binding that shadows a previous binding. These initialization statements take the form `x←x` for each binding `x` and ensure that each name referenced in a function will always resolve to the same frame throughout the entire body of the function. While other approaches are possible, using this method simplifies name look-up a fair bit. Visually, the following example and diagram illustrate the idea.

```
      {x←5 ⋄ {y←3 ⋄ {z←y+x ⋄ y←z+1 ⋄ x←y+y ⋄ x+x}⍬}⍬}⍬
36
```

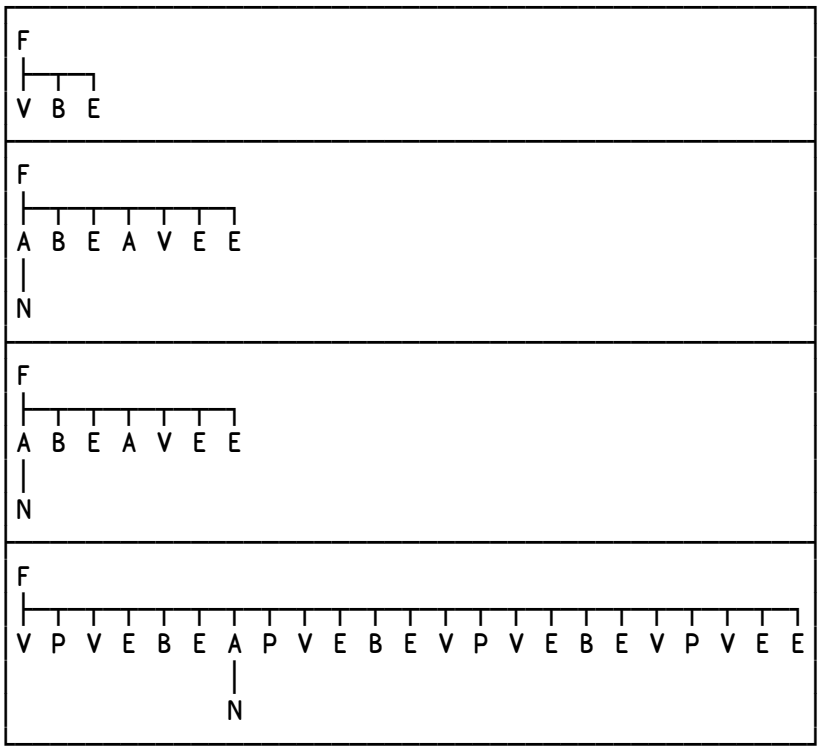
This example has 3 frames, with the final innermost function containing two free references, one to `x` and one to `y`. The following visualizes the name resolutions:



Notice that all the references resolve to frame 3, including the free references, since it is considered the nearest frame binding these names. The free references are accounted for by the implicit references

to the previous frames linking the bindings in frame 3 to bindings of the same name in frames 1 and 2.

Here is the above example in tree and column forms, including the newly generated `f` and `s` columns:




```
i 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19
p 0  0  0  0 44 44 44  4 44 44 44 44 45 45 12 45 45 45 45 45
t 3 10  1  2  0  1  2  7  0 10  2  2  0  1  7  2  0 10  2  2
k 0  1  1 -1  0  0 -1  0  0  1  1  0  0  0  0 -1  0  1  1  0
n 1 44 -5 -5  0 -6 -6 -7  0 45  0  0  0 -8 -9 -8  0 46  0  0
r 0  0  0  0 44 44 44 44 44 44 44 44 45 45 45 45 45 45 45 45

      20  21 22 23  24  25 26  27  28 29  30 31 32 33  34 35
      46  46 46 46  46  46 46  46  46 46  26 46 46 46  46 46
      10   9 10  2   1   2  0   9  10  2   7  1  2 10   9 10
       0   1  0  2   0  -1  0   1   0  2   0  0 -1  0   1  0
      -6 -11 -8  0 -10 -10  0 -11 -10  0 -12 -8 -8 -8 -11 -8
      46  46 46 46  46  46 46  46  46 46  46 46 46 46  46 46
```

```
        36 37 38 39  40 41 42 43 44 45 46
        46 46 46 46  46 46 46 46 44 45 46
         2  1  2 10   9 10  2  2  3  3  3
         2  0 ¯1  0   1  0  2  0  1  1  1
         0 ¯6 ¯6 ¯6 ¯11 ¯6  0  0  1  1  3
        46 46 46 46  46 46 46 46  0 44 45

        e
 0   ¯5
44   ¯6
45   ¯8
46 ¯10
46   ¯8
46   ¯6
        f
0 1 2 3 3 3 ¯1
        s
0 0 0 0 1 2 ¯1
```

Entering this pass a few structures are already set up. The `e` table contains a listing of all unique bindings, where the first column is the frame in which the binding appears and the second contains the name of the binding. The `b` vector contains all the binding node ids. The `rn` table contains all the binding locations including duplicates, with the same structure as `e`. The `f` and `s` vectors tally the same as `e` and map entries in `e` to frames and slots.

There are two major results of lexical resolution: firstly, all bindings associate with the appropriate slot that they mutate in their frame; secondly, each binding and name reference receives a slot and frame that they reference. In the case of bindings, the slot and frame point to the implicit binding used for references to the binding before it is bound in a frame. For name references, the frame and slot point to the location of the binding to which the name refers within the lexical stack. The bindings easily resolve to their slots in a single logical step but resolving references (binding and variable) requires searching up the lexical stack to locate the correct binding.

To begin this process requires a few more important values. The previous example continues to demonstrate the process. First compute the name references `v` and the combination of name and binding ids `y`. Name references are variable nodes that have symbolic names other than α, ω, αα, or ωω.

```
       n<¯4
0 0 1 1 0 1 1 1 0 0 0 0 0 1 1 1 0 0 0 0 1 1 1 0 1 1 0 1 1 0 1 1 1
    1 1 1 0 1 1 1 1 1 0 0 0 0 0
       t=10
0 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 1 0 1 0 0 0 0 0 1 0 0 0 0
    1 0 1 0 0 0 1 0 1 0 0 0 0 0
       ⍳(t=10)∧n<¯4
20 22 28 33 35 39 41
       v←⍳(t=10)∧n<¯4
       b
2 5 13 24 31 37
       v,b
20 22 28 33 35 39 41 2 5 13 24 31 37
       y←v,b
```

Next, save the names of bindings in `x` and give each binding a slot in its `n` field by looking up the binding in `e` and using the associated slot:

```
       y
20 22 28 33 35 39 41 2 5 13 24 31 37
       n[y]
¯6 ¯8 ¯10 ¯8 ¯8 ¯6 ¯6 ¯5 ¯6 ¯8 ¯10 ¯8 ¯6
       x←n[y]
       e
 0  ¯5
44  ¯6
45  ¯8
46 ¯10
46  ¯8
46  ¯6
       rn
 0  ¯5
44  ¯6
45  ¯8
46 ¯10
46  ¯8
46  ¯6
```

```
      eɩrn
0 1 2 3 4 5
      s
0 0 0 0 1 2 ¯1
      s[eɩrn]
0 0 0 0 1 2
      b
2 5 13 24 31 37
      n[b]←s[eɩrn]
```

This sets up the bindings to mutate the right slot. The next step resolves the references. Start with a buffer `i` to update with the index of the binding that matches the corresponding reference. This means `i` must be the same size as `x` and begins assuming each element is not yet found:

```
      (≠x)⍴c←≠e
6 6 6 6 6 6 6 6 6 6 6 6 6
      i←(≠x)⍴c←≠e
```

The search iterates over a transposed table of frame (row 0) and index into `x`. To begin with, the starting point for names is the id of the name itself and for bindings it is the enclosing frame:

```
      x
¯6 ¯8 ¯10 ¯8 ¯8 ¯6 ¯6 ¯5 ¯6 ¯8 ¯10 ¯8 ¯6
      ɩ≠x
0 1 2 3 4 5 6 7 8 9 10 11 12
      v
20 22 28 33 35 39 41
      r[b]
0 44 45 46 46 46
      v,r[b]
20 22 28 33 35 39 41 0 44 45 46 46 46
      (v,r[b])⌷⍨⊂ɩ≠x
20 2 28 33 35 39 41 0 44 45 46 46 46
 0 1  2  3  4  5  6 7  8  9 10 11 12
```

Why use the id `v` for names and `r[b]` for bindings? At each stage of search, the code uses these ids to look up the containing frame in `r`. Then it uses this frame to search for matching bindings. For names, searching begins in its local frame, but for bindings, if searching the local scope immediately matches the binding. Instead, the search for bindings must begin one frame up from local.

Within each iteration, first update the table with new frames. This updated table serves as the basis for the next iteration and is named `z`. In the iteration, `⍵` is the current iteration table.

```
      ⍵
20 22 28 33 35 39 41 0 44 45 46 46 46
 0  1  2  3  4  5  6 7  8  9 10 11 12
      r I@0⊢⍵
46 46 46 46 46 46 46 0 0 44 45 45 45
 0  1  2  3  4  5  6 7 8  9 10 11 12
      z←r I@0⊢⍵
```

Search by replacing the ids in row 1 with the appropriate names from `x` and then look up the references in `e`. Store the results in `i`:

```
      z
46 46 46 46 46 46 46 0 0 44 45 45 45
 0  1  2  3  4  5  6 7 8  9 10 11 12
      x
¯6 ¯8 ¯10 ¯8 ¯8 ¯6 ¯6 ¯5 ¯6 ¯8 ¯10 ¯8 ¯6
      x I@1⊢z
46 46  46 46 46 46 46  0  0 44  45 45 45
¯6 ¯8 ¯10 ¯8 ¯8 ¯6 ¯6 ¯5 ¯6 ¯8 ¯10 ¯8 ¯6
      ⌽x I@1⊢z
46  ¯6
46  ¯8
46 ¯10
46  ¯8
46  ¯8
46  ¯6
46  ¯6
 0  ¯5
 0  ¯6
44  ¯8
45 ¯10
45  ¯8
45  ¯6
      e
 0  ¯5
44  ¯6
45  ¯8
46 ¯10
46  ¯8
46  ¯6
      e⍳⌽x I@1⊢z
5 4 3 4 4 5 5 0 6 6 6 2 6
      i[1⌷z]←e⍳⌽x I@1⊢z
```

With the search at the current frame complete, proceed to the next iteration. However, unlike in some iterations used in prior passes, this one is computationally heavyweight. Thus, to avoid wasting search resources on references that we have found already, the code filters the iteration table to only those that have not been found yet. This gives the following iteration, the result of which is the next iteration table:

```
      ⍵
20 22 28 33 35 39 41 0 44 45 46 46 46
 0  1  2  3  4  5  6 7  8  9 10 11 12
      e⍳⍁x I@1⊢z
5 4 3 4 4 5 5 0 6 6 6 2 6
      z/⍤c=i[1⌷z]←e⍳⍁x I@1⊢z←r I@0⊢⍵
0 44 45 45
8  9 10 12
```

Assembled with the initial table gives the following:

```
      _←{z/⍤c=i[1⌷z]←e⍳⍁x I@1⊢z←r I@0⊢⍵}⍣≡(v,r[b]);⍁;⍳≠x
      i
5 4 3 4 4 5 5 0 6 6 6 2 1
```

At the completion of the above fixpoint the `i` buffer contains the search results and is a valid index vector into `f` and `s`. To complete the pass, recompute/transform the `f` and `s` vectors to be columns in the tree. Set the reference positions `y` to `f[i]` and `r[i]` and set everything else to `¯1`.

```
      ¯1⍴⍤≠r
¯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 ¯1 ¯1 ¯1 ¯1 ¯1 ¯1
      ¯1 ¯1 ¯1 ¯1 ¯1 ¯1 ¯1 ¯1 ¯1
      y
20 22 28 33 35 39 41 2 5 13 24 31 37
      '∘'@y⊢¯1⍴⍤≠r
¯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 ¯1 ¯1
      f[i]
3 3 3 3 3 3 3 0 ¯1 ¯1 ¯1 2 1
      s[i]
2 1 0 1 1 2 2 0 ¯1 ¯1 ¯1 0 0
```

```
      f[i]@y⊢¯1ρ⍤≠r
¯1 ¯1 0 ¯1 ¯1 ¯1 ¯1 ¯1 ¯1 ¯1 ¯1 ¯1 ¯1 ¯1 ¯1 ¯1 ¯1 ¯1 ¯1 3
      ¯1 3 ¯1 ¯1 ¯1 ¯1 ¯1 3 ¯1 ¯1 2 ¯1 3 ¯1 3 ¯1 1 ¯1 3 ¯1 3
      ¯1 ¯1 ¯1 ¯1 ¯1
      s[i]@y⊢¯1ρ⍤≠r
¯1 ¯1 0 ¯1 ¯1 ¯1 ¯1 ¯1 ¯1 ¯1 ¯1 ¯1 ¯1 ¯1 ¯1 ¯1 ¯1 ¯1 ¯1 2
      ¯1 1 ¯1 ¯1 ¯1 ¯1 ¯1 0 ¯1 ¯1 0 ¯1 1 ¯1 1 ¯1 0 ¯1 2 ¯1 2
      ¯1 ¯1 ¯1 ¯1 ¯1
      f←f[i]@y⊢¯1ρ⍤≠r
      s←s[i]@y⊢¯1ρ⍤≠r
```

The uses of ⍸ and ⍳ dominate the computational complexity of the pass. The initial definition of `v` is linear space and time with a logarithmic time critical path. The computation of `x` is linear time and space with a constant critical path. Setting `n[b]` is dominated by the search `e⍳rn`. The time complexity is `O((≠rn)×⊛≠e)` and linear space. The critical path is logarithmic in the size of `e`. The number of search steps in the main fixpoint is limited by the longest distance between a name's reference and its binding in terms of lexical frames. In cases where source may be expected to be strictly limited in lexical depth, these iterations may be thought of as bounded. Within each iteration of the fixpoint, the two limiting factors are the filters and search over `e`. On each iteration the critical path is always limited by the sum of the log of the size of `e` and the log of the size of the iteration table. The space is linear and the time is `O((≠ω)×⊛≠e)` where `ω` is the iteration table.

Here is the complete pass:

```
v←⍸(t=10)∧n<¯4 ⋄ x←n[y←v,b] ⋄ n[b]←s[e⍳rn] ⋄ i←(≠x)ρc←≠e
_←{z/⍤c=i[1⌷z]←e⍳⍥x I@1⊢z←r I@0⊢ω}⍣≡(v,r[b])⌸⍥⍳≠x
f s←(f s I⍤⊂i)⊣@y⍤⊂¯1ρ⍤≠r
```

Main CPU	Intel Xeon E5-2623v3 Haswell-EP 3.0 GHz Four Core 22nm CPU
Memory Type	DDR4-1866
L3 Cache	10MB
Boost Clock Speed	3.5 Ghz
Main Memory	32GB @ 1866Mhz (4×8GB DDR4 2133 MHz ECC)
Max CPU Memory Bandwidth	59 GB/s
GPU	GeForce GTX 980 PCI-e 3.0
Graphics Memory	4GB GDDR5
GPU Base Clock Speed	1126 Mhz
GPU Booz Clock Speed	1216 Mhz
GPU Memory Bandwidth	224.3 GB/sec
CUDA Cores	2048
Operation System	Microsoft Windows 10 1809 64-bit
Dyalog APL	APL/W-64 Version 17.0.36301
Racket Version	7.2
Haskell Version	TBD

Table 3. Benchmarking Test Machine Specifications

4. PERFORMANCE

The compiler described in section 3 was benchmarked against two implementations of the same compiler written using the Nanopass compiler framework, executed using the JIT compiled Racket and offline compiled Chez Scheme implementations. The Nanopass compiler framework represents a state-of-the-art approach to compiler construction in Lisp-like languages or other untyped, functionally oriented languages, with Racket representing one of the most popular of such implementations and the Chez Scheme compiler representing a system with a popular reputation of good performance on compiler and tree-manipulation benchmarks. The Co-dfns compiler was tested

with two computational devices (the main CPU and GPU of the benchmarking machine) on a constructed AST consisting of between `1019` nodes to `1019×2*14` nodes. Table 3 lists the test machine hardware specifications.

The two Co-dfns compiler implementations utilize the same code (described in Section 3) but execute on different architectures and with different runtimes. The Dyalog APL interpreter was used to run on the CPU, while a C++/ArrayFire compiled version with CUDA backend was used to execute on the GPU. The benchmarks below compare the Co-dfns compiler against timings taken for the equivalent data sizes on the CPU for the two Nanopass implementations. Timings consist of the average of 5 runs of the compiler recording total running time and all running times for the individual passes. For the Racket and Chez implementations, the benchmark initiates a garbage collection between each compiler pass to control for collection effects. These collections do not contribute to the timings, but collections that occurred during an executing compiler pass do contribute.

The AST for each data size was constructed by parsing and then replicating a typical APL program consisting of a variety of Black Scholes code snippets and various other small utility functions. The resulting unreplicated AST contains 1019 nodes, with a lexical depth of 3 and a tree depth of 15. Source code typically exhibits relatively shallow depth compared to the number of nodes, with relatively small lexical depth, and the benchmark APL program represents typical source code structures of this nature. To obtain larger data sizes, the 1019-node AST was replicated to create multiple namespaces and top-level values. The AST was duplicated in powers of 2, resulting in 15 derived ASTs consisting of `1019×2*N` nodes for each `N` in `[0,15)`. The Co-dfns compiler is agnostic

to the number of modules and thus, invocation on the entire set of modules in the AST was possible without modification. The Nanopass and Haskell implementations were both designed so that they received a set of modules to compile rather than a single module, thus allowing them to operate over multiple modules in the same way that the Co-dfns compiler was designed.

Understanding the relationship between nodes in an AST and the relative size of a given piece of source code depends on the density of that code. For example, for the data size 4 in the following benchmarks, this corresponds to approximately 16,000 nodes. In source code with the same relative node density as the APL program, this leads to a program of roughly 650 non-whitespace lines of code. However, when comparing this against the node density found in the Nanopass reference compiler, for example, this leads to approximately 1000 lines of code. This happens to correspond nearly to the size of the Nanopass reference compilers used in the benchmarks. Both the APL benchmark input source and the Nanopass Reference Compiler are included in the appendices, so comparing node densities of the reference compilers to the data sizes given in node counts leads to an intuition for the speedups in the following benchmarks relative to specific input sizes. Using the Nanopass reference compiler as an example, which is 1012 lines of code with a node per line density of roughly 14.5, this corresponds to a data size of roughly 3.8 in the benchmark graphs. Since most programmers tend to initially think in terms of lines of code, this relationship can assist in predicting the relative performance improvements of the compiler on specific input sizes in a more natural way than what the raw node count may provide.

The timing data in the following series of graphs uses a base-2 log scale in both dimensions. Each axis label represents a power of 2. Each graph indicates the relative speedup, calculated as the timing for the reference compiler divided by the timing for the Co-dfns compiler.

The reference compilers were designed to represent reasonable best efforts at canonical compiler design techniques. However, the overarching design of the compiler, in terms of the compiler passes, the intermediate representations (IRs), and the input language were all fixed. This allows for a direct comparison between the various tree processing techniques over the same manipulation and avoids these benchmarks becoming tests of differences in compiler architecture designs versus tree manipulation algorithms. That is, each compiler performs the same set of manipulations, over the same tree data types, in the same order, over the same input data. The only variation is the method of expressing the tree manipulations involved.

Attempts were explicitly made to represent the idiomatic tree manipulation methods that would be used in each reference framework. No attempts at “cleverness” were made with regards to optimizing any of the compilers, including the Co-dfns compiler. That is, the code avoids specialized implementations of primitives or non-idiomatic tricks leveraging knowledge about the domain in favor of generic techniques and runtimes. The coding standard does permit typical optimization procedures that do not negatively impact the idiomaticity of the source code, particularly with the traditional methods when they required some optimizations that deviated from norms to encourage better performance, including violating some data safety and type-assurances in order to make certain copying optimizations. The traditional compiler code also leverages additional data structures known

to have better behavior for our data sets because initial benchmarks indicated that some common or canonical techniques had astronomically poor performance. The graphs show only the optimized reference compiler timings; unoptimized versions were too slow for practical timing numbers. The respective sections below detail specific optimizations used for each compiler.

In addition to running time, memory usage was also generally observed. None of the systems used had an accessible method for obtaining precise memory usage behaviors in a way that did not drastically alter the behavior of the system, and so general monitoring was conducted instead for working memory usage. However, the memory requirements for the AST input into the compiler were measured more precisely on each platform.

4.1. AST Memory Usage for Co-dfns Compiler

Table 4 lists the AST sizes for the Co-dfns compiler compared to the Racket and Chez implementations. Whereas it is relatively more difficult to predict the memory usage of the ASTs for the reference compiler, a very tight bound on the memory usage for the Co-dfns compiler input AST representation is possible.

The inverted table AST representation input to the compiler utilizes four vector columns arranged in a 4-element vector. Each array will have some constant overhead factor `C` in addition to some constant per element size. The Dyalog APL interpreter attempts to "squeeze" any array down to the smallest representable element type. Most source code contains small depth, likely less than `2*8`,

Size	Dya	Rack	Chez	Size	Dya	Rack	Chez	Size	Dya	Rac	Chez
0	0.009	0.065	2.027	**5**	0.129	2.056	4.055	**10**	3.985	65.721	93.258
1	0.013	0.13	2.027	**6**	0.254	4.11	6.082	**11**	7.966	131.44	186.516
2	0.021	0.259	2.027	**7**	0.503	8.217	10.137	**12**	15.927	262.877	371.508
3	0.036	0.516	2.027	**8**	1	16.432	22.301	**13**	31.849	525.752	739.980
4	0.067	1.029	2.027	**9**	1.995	32.862	44.602	**14**	63.692	1051.502	1485.023

Table 4. Memory Usage in megabytes of Dyalog, Racket, and Chez Scheme input ASTs

meaning that the depth vector will need only 1 byte per element to store the tree structure. A similar situation exists for the type, kind, and name vectors. The type and kind fields both are small range enumeration, thus the interpreter will also represent them with a single byte per element. The parser converts the name vector from a vector of strings to a vector of symbols and an explicit symbol table before passing the AST to the compiler, so the minimum size per element in the name vector is the size necessary to index into the symbol table. The symbol table is also likely to be small, so a single byte often suffices, and this is the case for the code used in the benchmarks. Thus, the per node cost in terms of memory requirements for the parsed AST is 4 bytes per node.

Accounting for a pointer for each column stored in the inverted table, this gives an estimate of the size of the AST of `(5×C)+32+4×N` where `N` is the number of nodes and `C` is the array header overhead. Plugging in appropriate values to this expression and comparing them to the actual memory requirements given in the above table will show just how close this estimate is, though the above does not take into account the size of the symbol table, which is included in the memory requirement measurements above.

The above table illustrates the significant differences between the space usage of the Nanopass implementations and the Dyalog APL implementation. These differences closely relate to the performance of the Chez Scheme and Racket Nanopass implementations as discussed in the corresponding sections.

4.2. GPU Data Transfer Overheads

GPU timings in the following sections are based entirely on the cost of executing the code after the data is transferred to the GPU memory and before that memory is transferred back. Before the data is transferred to the GPU, the AST received from the parser is replicated according to the number of nodes required for the given data size so that there is no data sharing between sub-trees in the AST. This ensures that the AST represents a fair and complete representation of the actual memory costs associated with a tree of that size. The memory usage given in the previous section indicates the amount of memory required for the AST after replication. This entire tree is then transferred to the GPU for the benchmark. The transfer times for each data size pre-pass and post-pass are as follows (times in milliseconds):

	1k	**2k**	**4k**	**8k**	**16k**	**32k**	**64k**	**128k**	**256k**	**512k**	**1m**	**2m**	**4m**	**8m**	**16m**
Pre	0.3	0.3	0.3	0.3	0.3	0.4	0.4	0.5	0.7	1.1	1.9	3.3	6.4	12.9	25.1
Post	0.8	0.8	0.7	0.7	0.8	0.9	1.0	1.4	3.5	6.3	12.8	24.6	47.9	94.5	185.9

For the smallest benchmark sizes, the transfer overheads remain relatively constant until around the 32k mark.

Data transfer overheads for a given machine can limit the total potential parallel speedup for any given system. It thus behooves any implementor to keep as many things on the GPU for as long

as possible and to compute as much on the GPU as possible to avoid the overheads of transferring data between the CPU and GPU memories. Some systems eliminate or reduce these overheads by sharing main memory between CPU and GPU architectures, but the relative parallel speedup available by leveraging GPU acceleration for these transformations will be limited by the number and cost of data transfers that are utilized in the overall application workflow.

In the case of compilers, however, note that this work focuses on one aspect of compiler design, namely, the issue of tree transformations. For sophisticated compilers, many other, potentially expensive analysis passes or parsing passes may occur intermingled with the tree transformations. Previously, these passes have been accelerated on the GPU (Prabhu et al. 2011) but required transferring information back to the CPU to continue compilation. This work enables potentially the entire compiler to appear on the GPU, from parsing to code generation, when combined with existing GPU algorithms for optimization, analysis, parsing, and code generation. This helps to eliminate the need for data transfer overheads across the entire compiler, even if the data transfer overheads may potentially limit available parallelism for these specific operations in isolation.

4.3. CPU vs. GPU Performance

Figure 1 and Figure 2 graph the performance of the Co-dfns compiler executed on the CPU versus the GPU. The implementations use the same source code but not the same runtime. The CPU timings used the Dyalog APL interpreter runtime, while the GPU times utilize a compiled version of the Co-dfns source built on the CUDA backend using C++/ArrayFire. The timings show consistent scaling, but the Dyalog APL interpreter leverages dynamic optimizations at runtime which helps to improve

performance of the CPU results at certain data sizes, resulting in higher variation at certain sizes. The average GPU speedup over the CPU implementation at the highest data size was around 6 times. This generally corresponds to the relative differences between the memory bandwidth seen on the CPU and the GPU, lending credence to the claim that these operations are primarily memory bound operations. When compared to more traditional implementations, the relative similarities in speed between the two Co-dfns implementations versus the traditional implementations also suggests that the Co-dfns implementation makes more efficient use of the available CPU memory bandwidth.

The CPU based implementation uses both fully iterative code as well as vectorized code during execution, but it does not leverage any multi-threading or multi-core operations. In addition, the structure of the source enables the interpreter to dynamically leverage more optimal data structures when working on the data and more optimal algorithms than might at first seem obvious given the array-based approach. In particular, the interpreter may utilize on demand hash tables, vectorized binary search, vectorized arithmetic, and optimized array resizing/caching in order to improve performance of the code. Aside from utilizing some documented idiomatic structures for improved idiom recognition, taking advantage of these optimizations on the interpreter does not require the use of any major adaptations of the source code, nor changing of the data types or primitives used.

The GPU based implementation does not benefit from nearly the same level of runtime or compile-time optimization or tuning. Indeed, aside from utilizing reasonable core algorithms for things like scan, as well as function fusion for scalar and reduction operations, the GPU compiled

implementation contains almost no optimization. Some core operations, such as binary search, utilize asymptotically sound approaches but without the expected fusion and caching optimizations possible when using CUDA's three major levels of parallel composition, namely, block, warp, and thread-level abstractions. This results in critical paths and constant time overheads potentially much greater than they could be given the extensive performance difference sometimes seen between unoptimized and optimized GPU programs. Most of the algorithms utilize unnecessary numbers of kernels that increase the critical path and global memory bandwidth usage more than would be strictly necessary. Significant potential optimization opportunities exist for the GPU implementation. In short, the GPU implementation is sound but relatively inefficient by current GPU optimization standards.

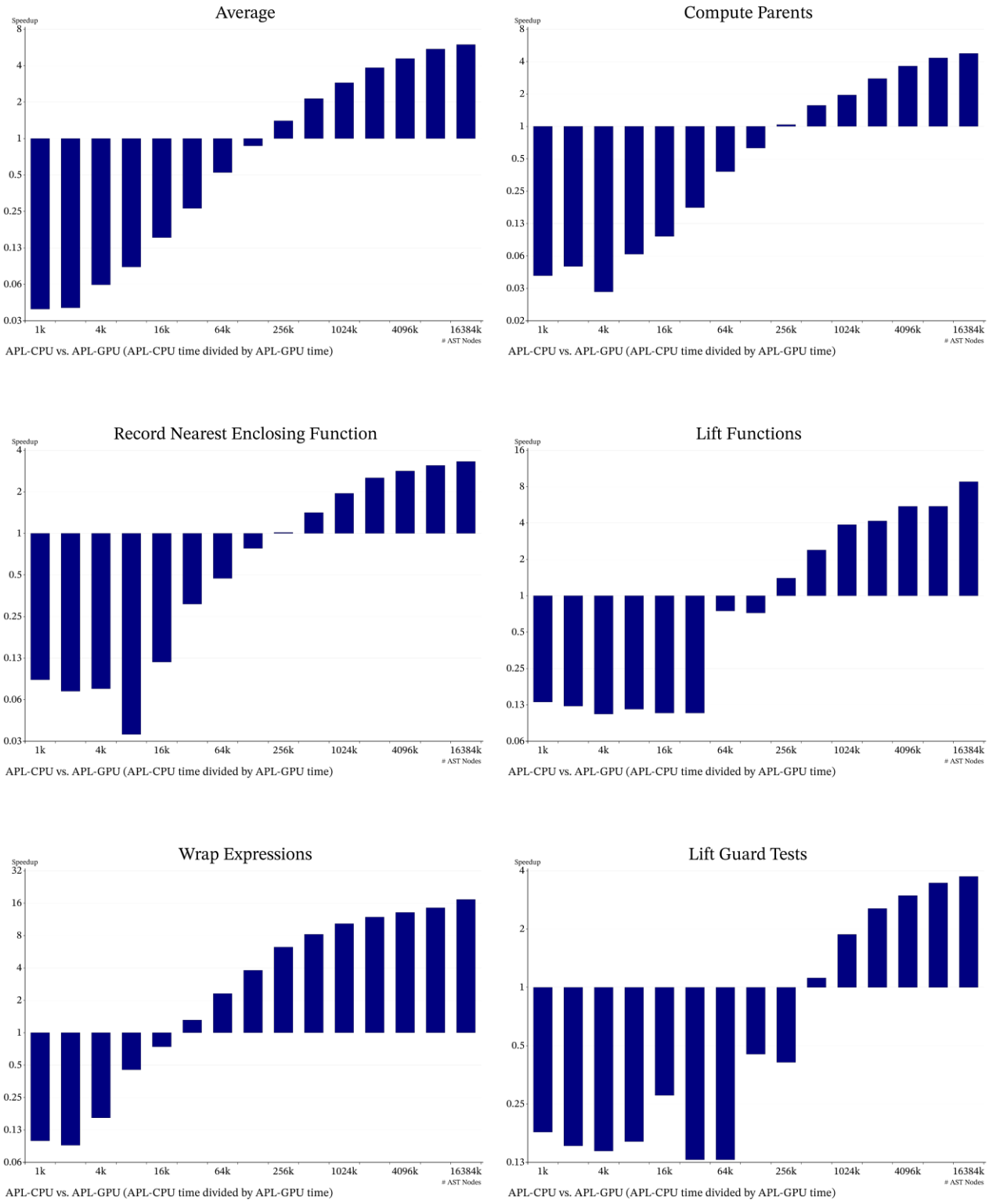
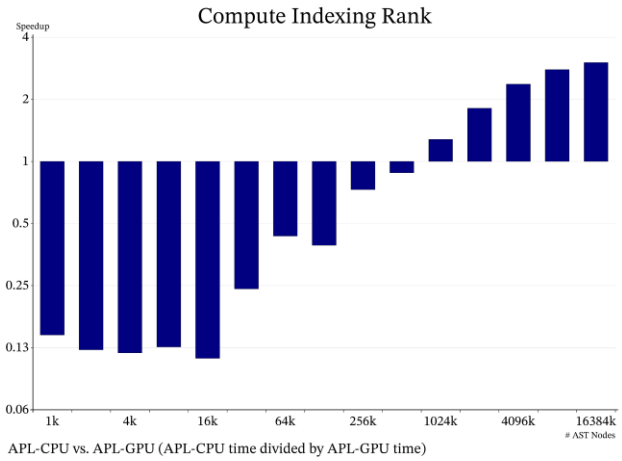
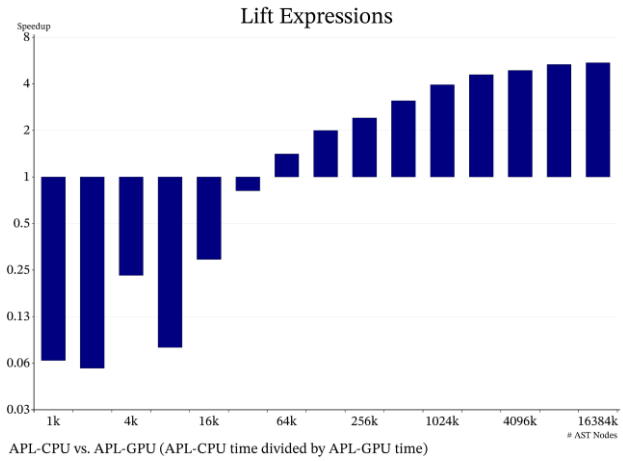
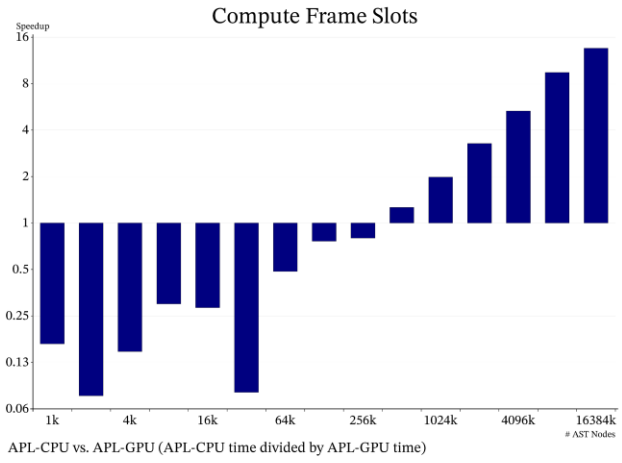
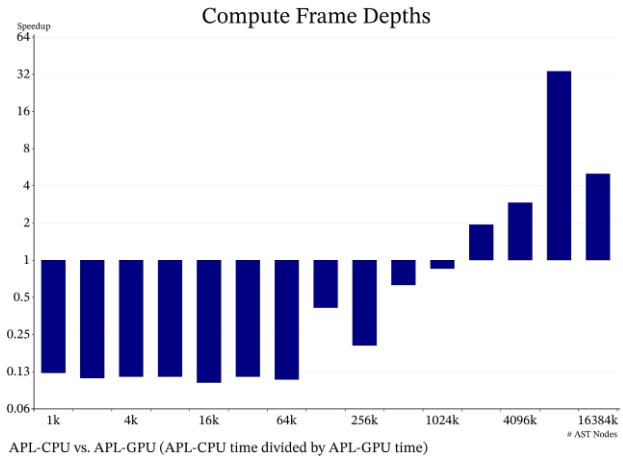


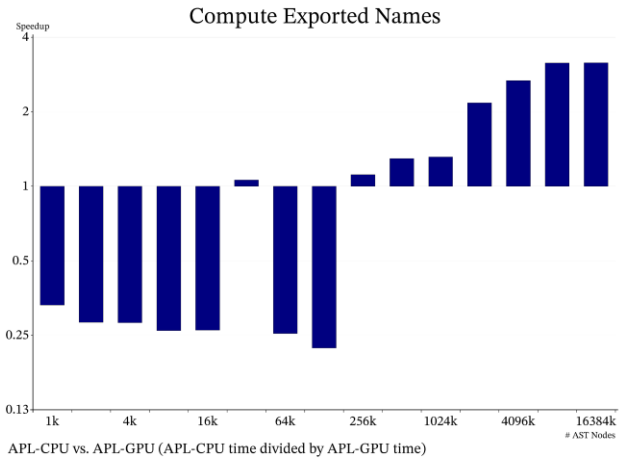
Figure 1. CPU vs. GPU Performance for Co-dfns Compiler











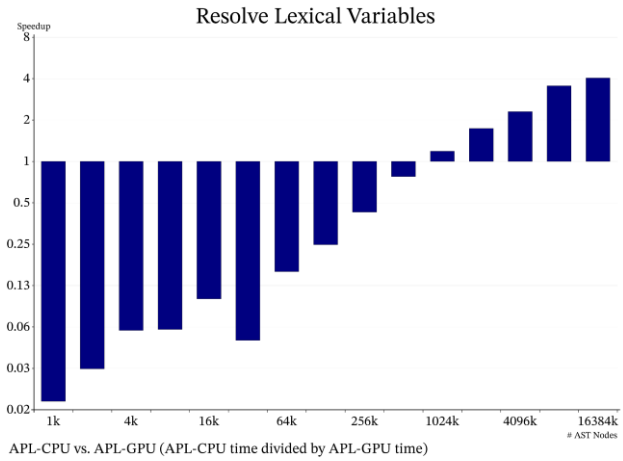


Figure 2. CPU vs. GPU Performance for Co-dfns Compiler, continued.

As a result, in cases where the operations have relatively small critical paths but very large memory manipulation requirements, such as with Expression Lifting or Expression Wrapping, the GPU's higher memory throughput and better computational parallelism (where function fusion can play a part) quickly outperforms the CPU, often by a fairly wide margin (more than 17x in the case of expression wrapping). However, on passes with longer critical paths where lookup and search are critical elements, the simplistic and unoptimized GPU algorithms require more data to work with before they begin to compete with the highly tuned and optimized runtime environment of Dyalog APL. The Lexical Resolution pass demonstrates this clearly. Note, however, that in all cases the GPU eventually outperforms the CPU as the data grows, and as the size grows larger, no single pass sees less than a 3x performance improvement at scale over the CPU. Improvements to the GPU implementation and optimization would undoubtedly yield lower constant factors and thus enable the GPU version to see improvements over the CPU runtime at smaller data sizes, but the current results indicate that the parallel scaling as size increases for the GPU algorithms agrees with the theoretical behavior of these algorithms, even in the presence of algorithmic tuning on the CPU side otherwise lacking on the GPU side.

	Lines of Code	**Tokens**	**Unique Names**	**Nodes in AST**
Nanopass	1012	20947	248	14680
Co-dfns	17	760	74	948

Table 5. Comparison of the Nanopass Reference Compiler source code metrics vs. Co-dfns

4.4. Performance vs. Racket Nanopass Implementation

Appendix C lists the Nanopass reference compiler source on page 211. Table 5 includes a comparison of the source code complexity between the Racket reference compiler and the Co-dfns compiler. Figure 3 through Figure 6 graph the relative speedups between the Racket reference compiler and the Co-dfns CPU and GPU implementations.

On the CPU, the Co-dfns compiler consistently outperforms the Racket reference implementation for all data sizes in all but two specifically optimized analysis passes. The GPU implementation also scales well and demonstrates consistent speedups even at very low data sizes. On average, these speedups are anywhere from 33x to 200x on the CPU and GPU at large sizes. On individual passes, the speedups exceed 1500x and beyond. Given the memory-bound nature of these algorithms, and the relative memory bandwidth between the CPU and GPU on the test machine given in Table 3, the very large speedups not only seem excessive, but somewhat incredible. What is going on? A careful study of the various aspects of the Nanopass framework and comparing the underlying implementation requirements between it and the Co-dfns compiler design reveals that, indeed, these benchmarks are testing Racket memory subsystem, but that the traditional design is significantly worse than the Co-dfns compiler architecture than it might first appear. Multiple factors contribute to this, and it is worthwhile to carefully consider each.

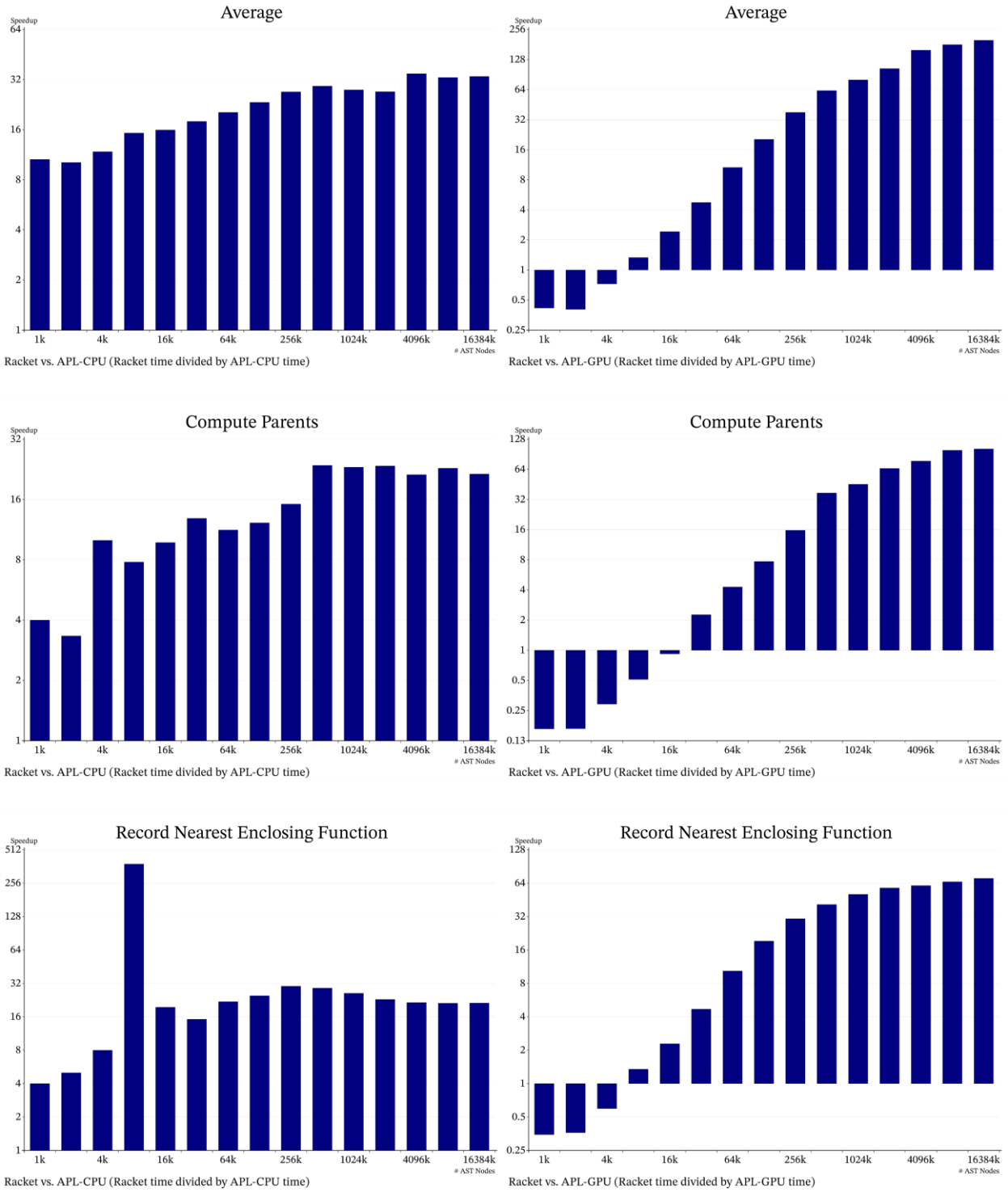


Figure 3. Average Speedup vs. Racket Nanopass for CPU (Left) and GPU (Right) platforms. Vertical Axis: Speedup. Horizontal Axis: Size of AST in nodes.

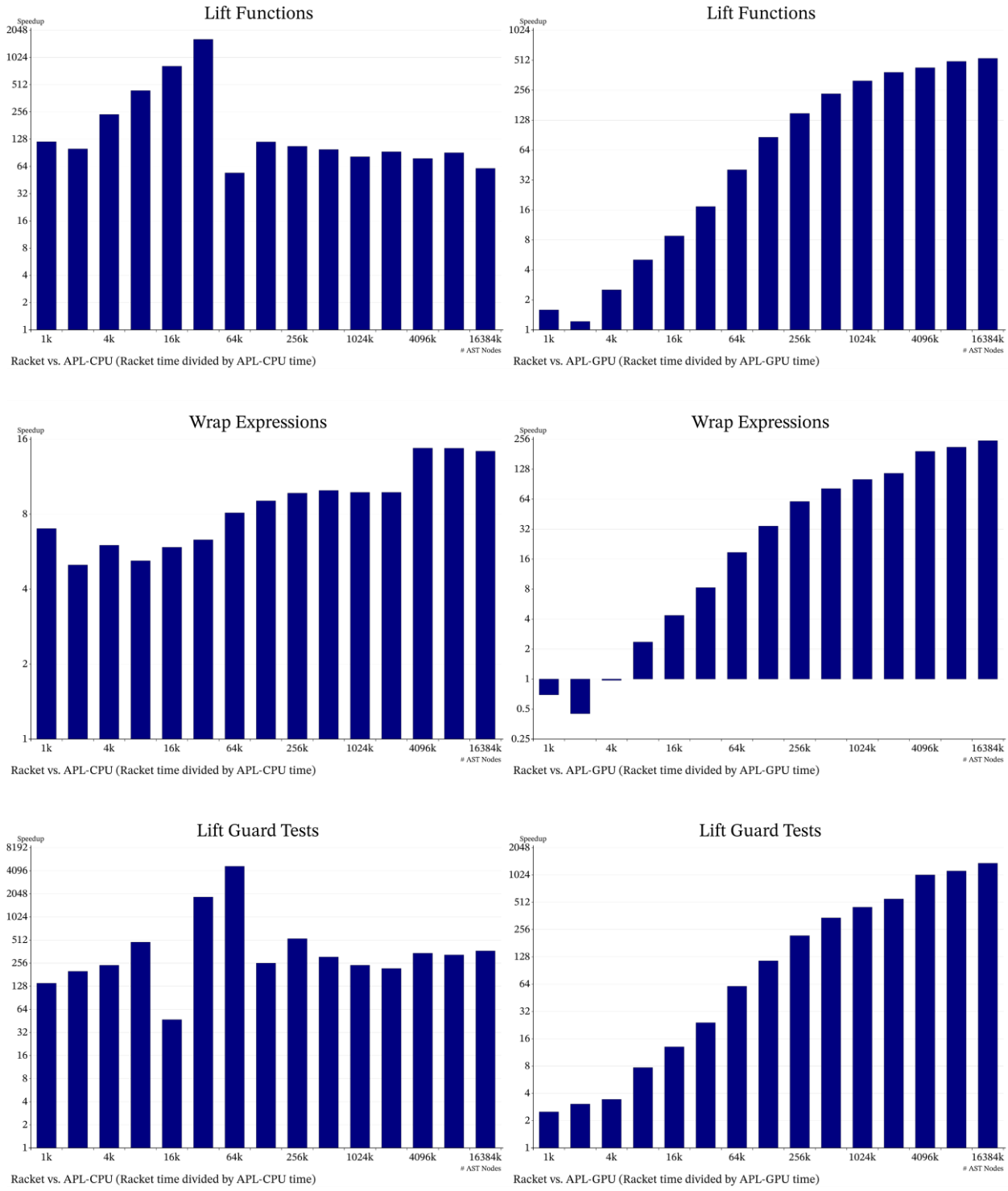


Figure 4. Speedups vs. Racket Nanopass for CPU (Left) and GPU (Right) platforms, continued.

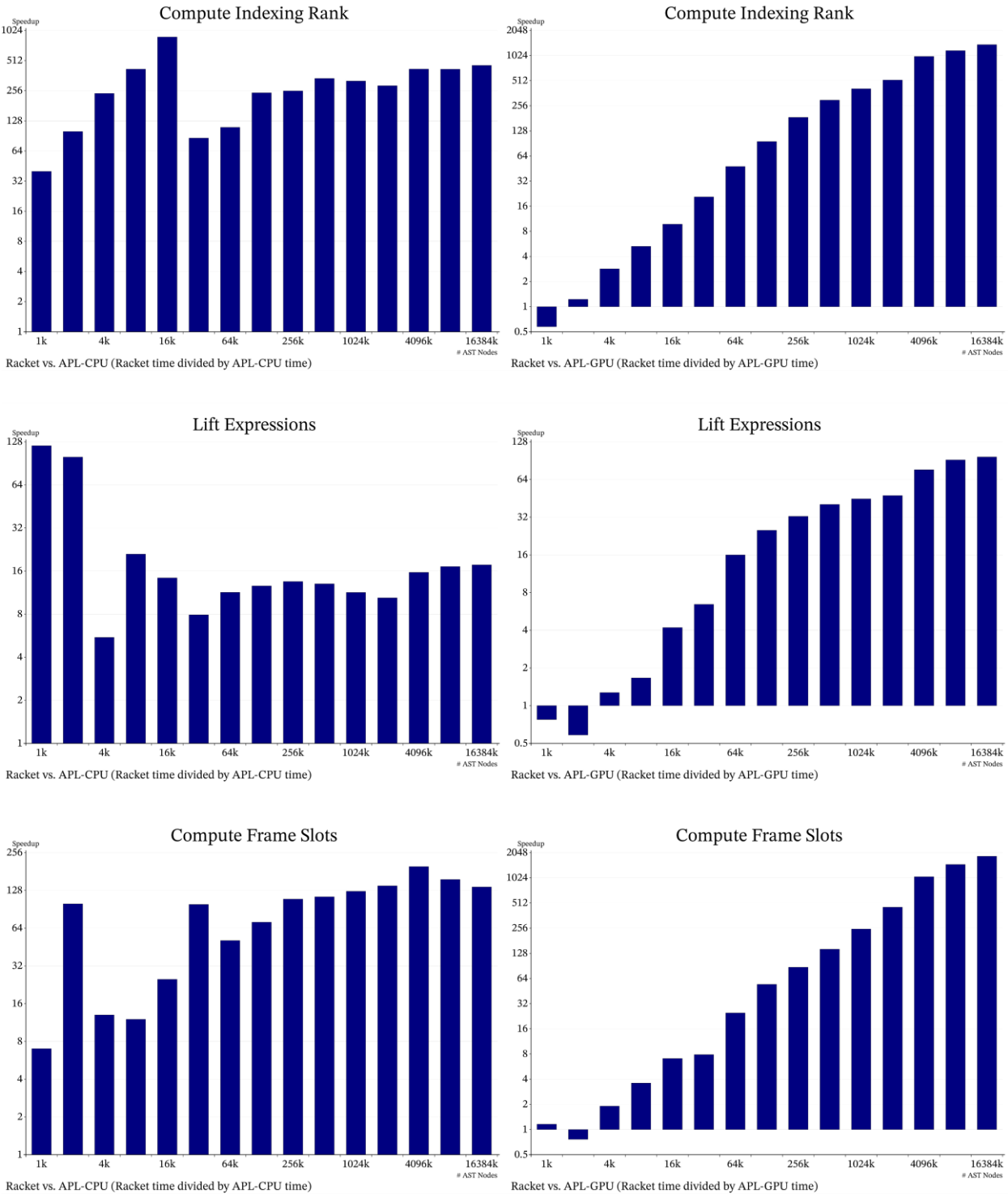


Figure 5. Speedups vs. Racket Nanopass for CPU (Left) and GPU (Right) platforms, continued.

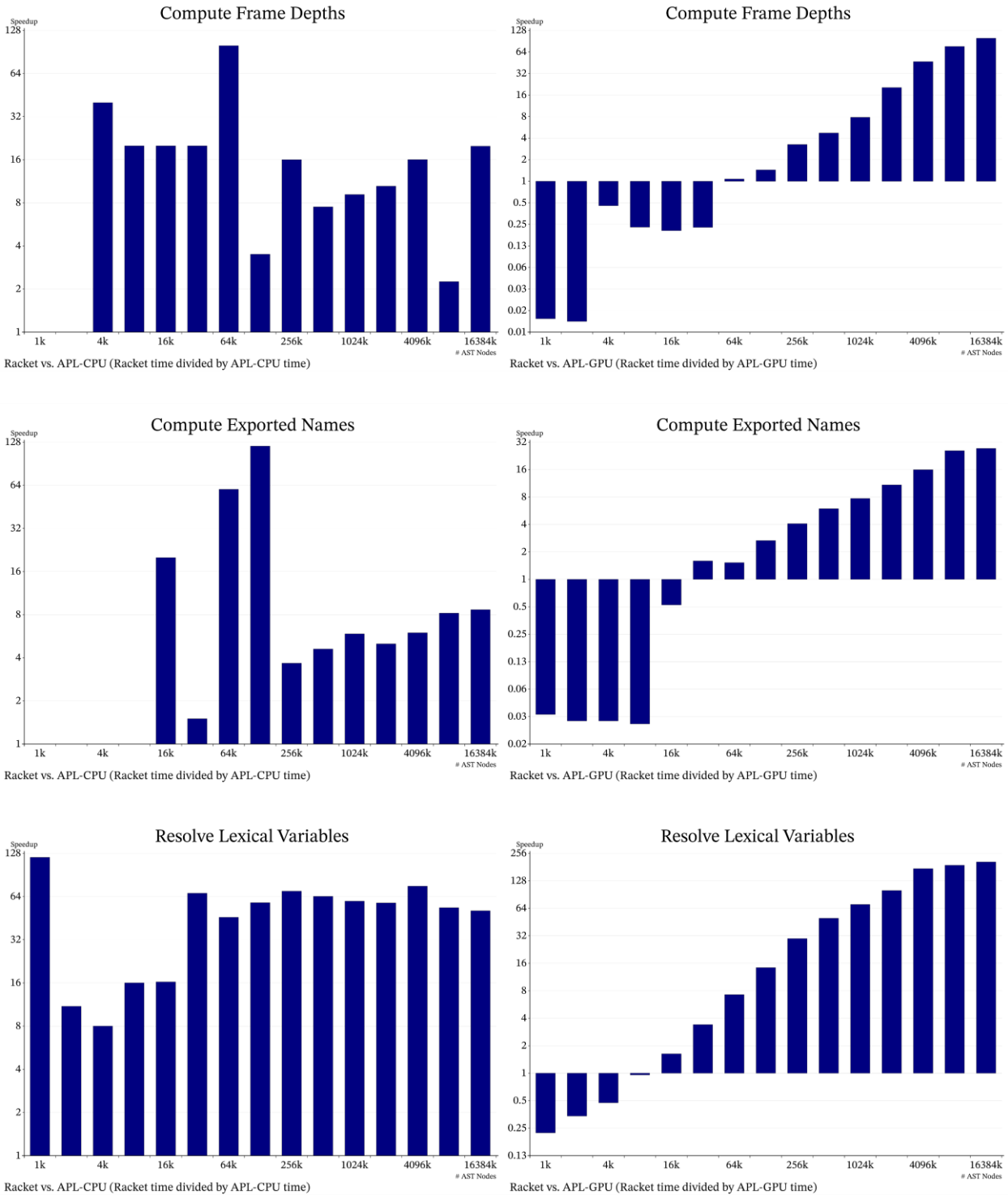


Figure 6. Speedups vs. Racket Nanopass for CPU (Left) and GPU (Right) platforms, continued.

The first indicator of just how different the two sources are in practice is the load times. The Dyalog APL interpreter loads the compiler instantly. However, Racket requires approximately 19 seconds to load and construct the requisite 1000 lines of code on a plain JIT Racket session (DrRacket, the Racket IDE, requires significantly more time due to orchestration). The loaded implementation in Dyalog requires almost no additional memory over the baseline, whereas in Racket, the memory usage of the system goes from 71MB at the start to 204MB after loading the implementation. The DSL design of the Nanopass framework contributes to load time since the Nanopass code must expand into core Racket code. During expansion, Racket not only constructs the underlying data types for the compiler's ASTs, but also type checks transformations and pattern matching clauses. The additional record type instrumentation and construction, and type checking patterns against those types, while arguably a significant programming convenience, does increase code size and load time. The large amount of generated code to execute can negatively affect code caching and execution performance.

Next, the two systems utilize very different AST representations. The Nanopass framework utilizes a typical record system to construct the AST, found in most compiler designs. Section 3 discusses this record-oriented approach in some detail at a theoretical level, but examining the differences in memory usage between the AST used in Dyalog APL and the AST used in the Nanopass implementation (Table 4) reveals the stark contrast between the two in practice. At data size 14, the APL implementation requires 63MB of memory to store the AST as it receives it from the parser, whereas the same AST in Nanopass consumes 1GB. Moreover, the APL implementation arranges the AST into 4 contiguously allocated memory regions, each corresponding to a single field/column in the

AST. This results in a high degree of memory locality. When combined with the data-parallel primitives utilized by the Co-dfns compiler, a very high number of memory accesses found in the compiler match a sequential or strided pattern rather than arbitrary patterns. On the other hand, the AST structure of the record-oriented approach found in the Nanopass compiler is record-local, meaning that each record is likely to be contiguously allocated, but on the whole, each record resides in a location difficult to predict with any accuracy that will not result in sequential access patterns. Moreover, most pattern matching implementations will load the entire record of any node to extract relevant data and construct the new node after recursion, thereby increasing the amount of reads and writes from main memory. Unlike the sequential access guaranteed by the Co-dfns arrangement, the lack of contiguous allocation combined with the pattern matching implementation results in a greater total of reads and writes, none of which likely match a sequential access pattern. The Co-dfns design separates the fields from each other and arranges fields contiguously with one another, enabling read and write over a single field without touching the memory for the rest of the record structure. The Co-dfns design utilizes very few allocated regions, so memory can be freed and allocated in large chunks with little to no collection overheads. In summary, the Co-dfns representation takes advantage of the following:

1. Contiguously allocated fields
2. Sequential field local access patterns
3. Greatly reduced storage requirements per node
4. Very low, near constant time collection requirements

The Co-dfns compiler traverses less memory, does not need to work as hard to collect that memory, traverses that memory using sequential access patterns instead of random patterns, and makes heavy use of vectorized computations over small sized data since the element sizes are small.

As an example, in order to compute parent nodes, the Co-dfns compiler need only work with the depth vector, which is $1/4^{th}$ the total size of the AST's memory requirements, in the case of a 63MB AST, roughly 16MB of data. That data is all laid out in a contiguous way and the values can be accessed sequentially in optimized fashion, and because each element is only a single byte, working with these values lends itself well to the small element size vector instructions found on both the GPU and the CPU. On the other hand, the Nanopass compiler may read over the entire 1GB AST, since the Nanopass framework does not optimize what elements of the record it needs to read and what it doesn't; it does not guarantee a sequential access pattern; and, the pointer-heavy random-access structure means loading in large, 64-bit values to read and work over, instead of the much smaller 8-bit values, making vectorization more difficult and less efficient, to the point that the Nanopass framework simply does no vectorization.

The Nanopass framework also incurs an inherent design cost. The Co-dfns compiler does not embed structural checking into the code, requiring the programmer to insert any higher-level type safety checks manually and explicitly into the code. Nanopass, on the other hand, introduces several safety checks both at the compile and run time to ensure that the structures that one creates inside of the Nanopass framework are type safe with respect to the language definitions. Much of this checking occurs at the compile time and does not contribute to the run time overhead of the resulting compiler.

However, the underlying Racket system cannot push all of this to compile time so some checks must occur at run time. The Dyalog APL system is an untyped interpreter (vs. JIT compiled for Racket) with a number of type safety checks but the array-oriented model allows many of these checks to occur at an array level and often with better memory efficiency, rather than the node level of the AST, reducing their cost to the overall performance at run time. The run time checks conducted by the Nanopass system require more sophistication and occur at the node level in a way that is less cache and memory friendly, even discounting the additional semantic level checks provided by Nanopass.

Unfortunately, it is not just the large memory usage of the AST, nor its random-access pattern, nor the extra costs of type safety that cause problems, but the functional programming nature of the Nanopass design and limitations built into the framework. The Nanopass framework emphasizes the construction of small IRs that minimally differ from each other, using small, easily defined "nano passes" to transform one IR into another, eventually leading to the necessary end target. It encourages functional programming principles, meaning that generally, the programmer does not mutate one tree into another, but instead generates a new tree from a prior one. This has several safety and usability benefits. Prior research demonstrates the use of Nanopass to construct fast compilers even when copying the tree many times throughout the compiler, since the computational complexity of certain passes often dominates these copy overheads. Nanopass provides no canonical mutative style, and languages that are defined do not share common sub-tree structures at the type level. If the structure of the AST changes enough to warrant a new language, with the main aesthetic encouraging many little languages, rather than an overly complex and general set of fewer IRs, which would reduce the

type-checking, convenience, and safety features of Nanopass, even if the operation only works over a few elements of the tree, sub-trees in one language are copied into isomorphic sub-trees of the new language type and they are not shared.

This record-oriented, functional approach to tree manipulation, construction, and traversal pervades compiler and language design so strongly that most lack syntactic or type-level support for working with trees structured in any other way, necessitating significant effort for the programmer to use any other method. This restriction, generally accepted, comes at a large performance cost when compared against the Co-dfns architecture. Specifically, no facility exists to easily modify the record structure in a way that does not recopy the entire tree in order to add or remove structural data from the tree. For example, to remove a specific child from a node, or add a new field for some additional information, or anything that changes the record type for a given node type, it generally necessitates an entirely new structure (language in Nanopass's terminology) that may require copying the entire tree, or very large components of it.

In the set of passes used in the Co-dfns compiler, this happens at nearly every pass, thus necessitating a full copy of the AST on nearly every pass of the compiler when translated into the Nanopass reference compiler. The Co-dfns compiler separates each field into its own space and does not enforce a strict type-discipline on the node types of the AST. This combination reduces the amount of type-safety and checking implicit within the design (though such safety can be added back into the design explicitly), but it also means that adding or deleting fields, changing their structure, or altering the parent-child relationships between the various nodes comes at no cost beyond the information to

add or remove. Adding a new field to every node in the AST does not require any modification or even any reading of the existing tree.

Even though the Co-dfns compiler takes this less type-safe approach, it still leverages a dataflow, spiritually functional approach, just in a way that enables bulk operations to transform the tree without unnecessary copying.

This advantage is large enough that the benchmarked Nanopass compilers incorporate optimizations to ameliorate this disadvantage where possible. At the end of the compiler, three "analysis" passes compute the frames, slots, and exports used by the final lexical resolution pass. Since these passes only extract information from the AST into separate "databases," in theory each pass can reuse the AST in its entirety. Unfortunately, the Nanopass framework has no way of permitting this AST sharing between languages, so to enable AST sharing the Nanopass compiler defines a single language used by all four passes at the end of the compiler. This allows for AST sharing between the last four passes, but it still necessitates a single new copy at the beginning of these four passes. The first computational pass at the end highlights the performance cost with copying versus the following two computational passes that share the AST. Note that these two passes, with this optimization, are the only two passes able to achieve any level of performance parity with the CPU execution of the Co-dfns compiler for small data sizes. This shows two things: firstly, the cost of excessive copying greatly affects performance on these sorts of transformations; and secondly, copying alone does not entirely account for the scaling gap, as it was not enough to scale beyond the Co-dfns compiler.

The performance of the Racket compiler demonstrates a compounding set of negative performance factors that all exacerbate each other, leading to vastly lower than expected performance. Some of these issues may arguably be considered inherent to the nature of the traditional tree manipulation algorithmic techniques. In particular, the heavy reliance on record-oriented, random-access, garbage collected, pointer-heavy structures is a major roadblock to efficient manipulation, as is the inherent difficulty in efficiently vectorizing recursively defined, top-down algorithms. However, some changes in implementation support could immediately improve performance, such as enabling two languages to share sub-tree types to avoid unnecessary copying.

Efficient linearizations for record-oriented ASTs could also have positive effects on the traversal speed for the traversals that were optimized for the linearization, but it would be difficult to maintain the abstraction level generally expected from such frameworks while still achieving good traversal performance over a wide array of traversal patterns, since effectively doing this might necessitate exposing underlying implementation details that were previously assumed to be desirably hidden. Such linearizations are also unlikely to net a complete amelioration of the copying problems for transformation algorithms.

4.5. Performance vs. Chez Scheme

The Chez Scheme Nanopass compiler found in Appendix D corresponds closely to the Racket source. The relative complexity of the code remains very close to that of the Racket source found in Table 5. Figure 7 through Figure 10 graph the relative speedups between the Chez Scheme Nanopass source

and the Co-dfns CPU and GPU implementations. The source code for Chez Scheme was compiled using `(optimize-level 2)` with Chez Scheme version 9.5.2 in 64-bit mode.

The Chez Scheme compiler performance mirrors that of the Racket system, but usually with some constant factor overall improvement. In particular, the very long load times noted for the Racket system were not present in the Chez Scheme system, with the Chez Scheme system requiring only a second or two in order to load the source versus the larger times for the Racket system. This was still noticeably longer than the load times involved for the Dyalog system, but not nearly as pronounced.

The overall execution timings for Chez Scheme improved over the Racket version usually by between 2 and 4 times. At scale, the Dyalog APL interpreted implementation of the Co-dfns compiler saw an average speedup of 9.4x versus the 33x seen in the Racket source. On the GPU, the speedup on average at scale was 56.1x. This shows a factor of between 3 and 4 improvement over the Racket results. This appears to come mainly in the form of constant factors, as the overall behavior of the speeds across different AST sizes appears to scale proportionally. In particular, the CPU speedups versus Chez Scheme continue to show large constant speedups across the range of data sizes, while the GPU sizes continue to show growth in speedups as the data sizes grow larger.

The Chez Scheme implementation has two distinctive behaviors over the Racket implementations. The performance on small data tends to be much better than that of the Racket system, and it matches or outperforms the Dyalog interpreter at small sizes sometimes by reasonably large factors, whereas the Racket system was never able to do this. Moreover, the Chez Scheme system tends to utilize more memory than the Racket system but perform better at scale. This would suggest

that the garbage collection and allocation performance of the Chez Scheme system is significantly better than that of the Racket system.

Thus, the Chez Scheme system appears significantly more efficient than the Racket system for this source, which is not surprising given Chez Scheme's history. However, the underlying limitations still present scalability issues for Chez Scheme; both the Dyalog CPU implementation, which is interpreted, and the GPU implementation of Co-dfns outperform the Chez Scheme implementation quickly while suffering relatively little at small sizes.

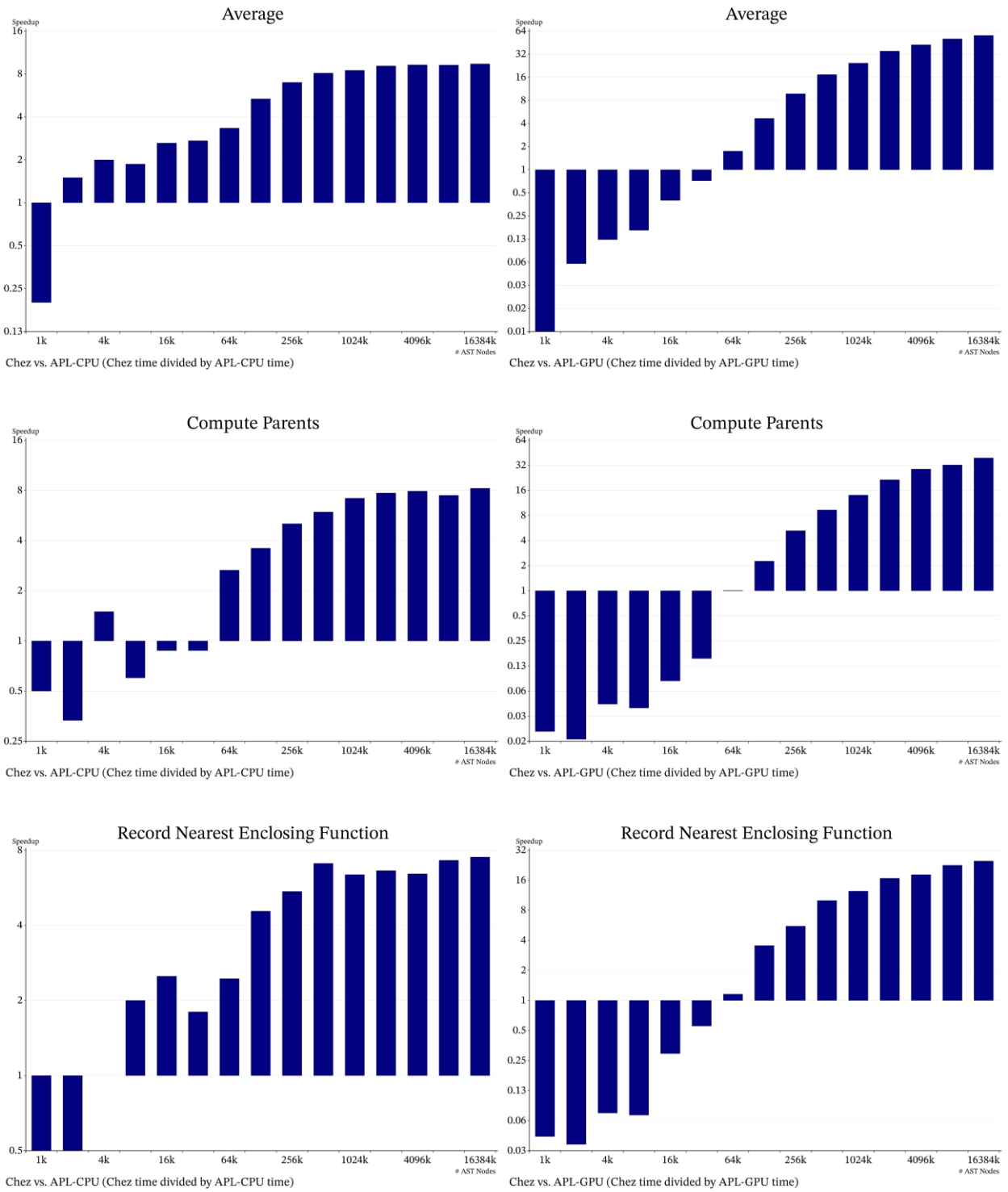


Figure 7. Average Speedup vs. Chez Nanopass for CPU (Left) and GPU (Right) platforms. Vertical Axis: Speedup. Horizontal Axis: Size of AST in nodes.

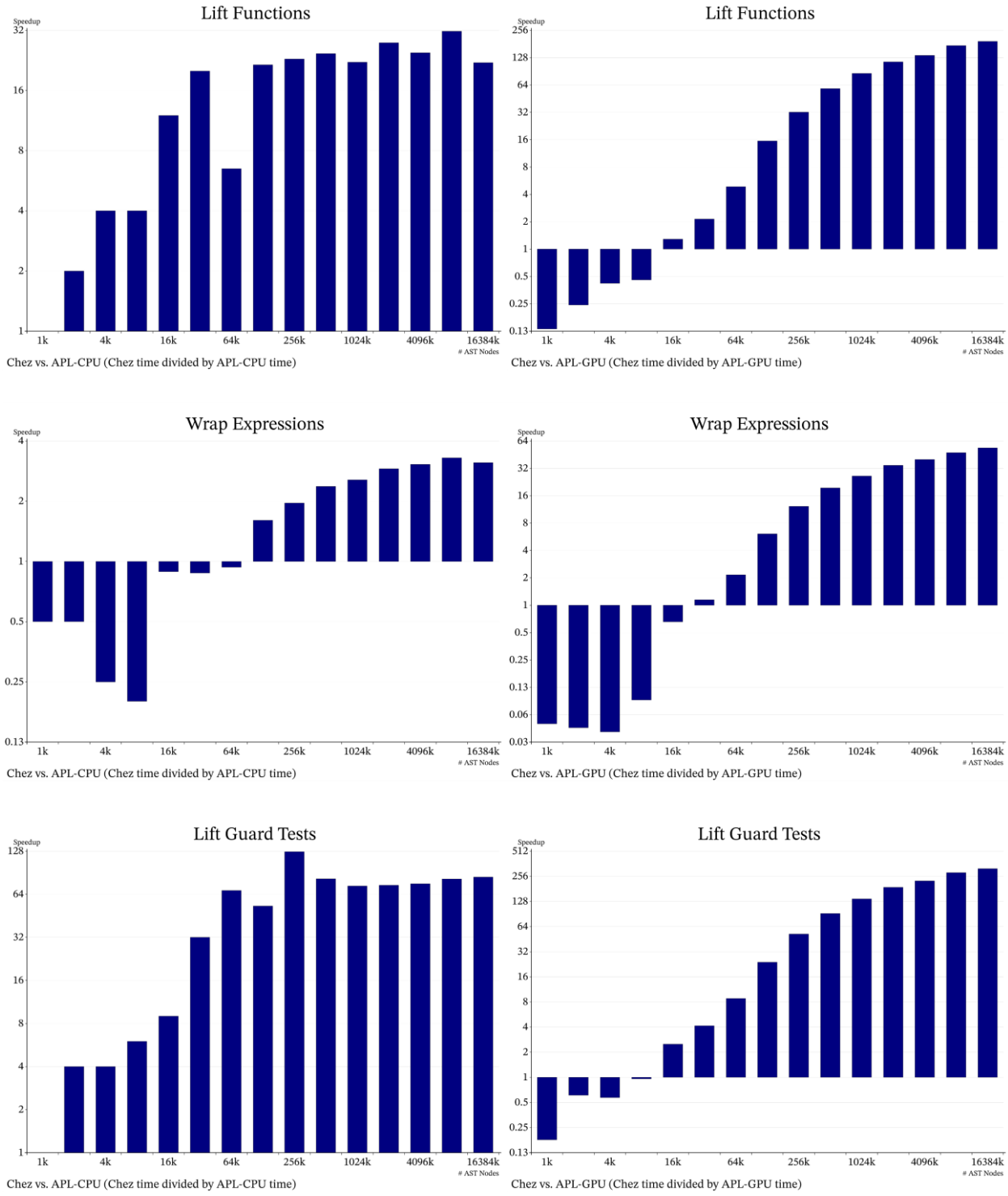


Figure 8. Speedups vs. Chez Nanopass for CPU (Left) and GPU (Right) platforms, continued.

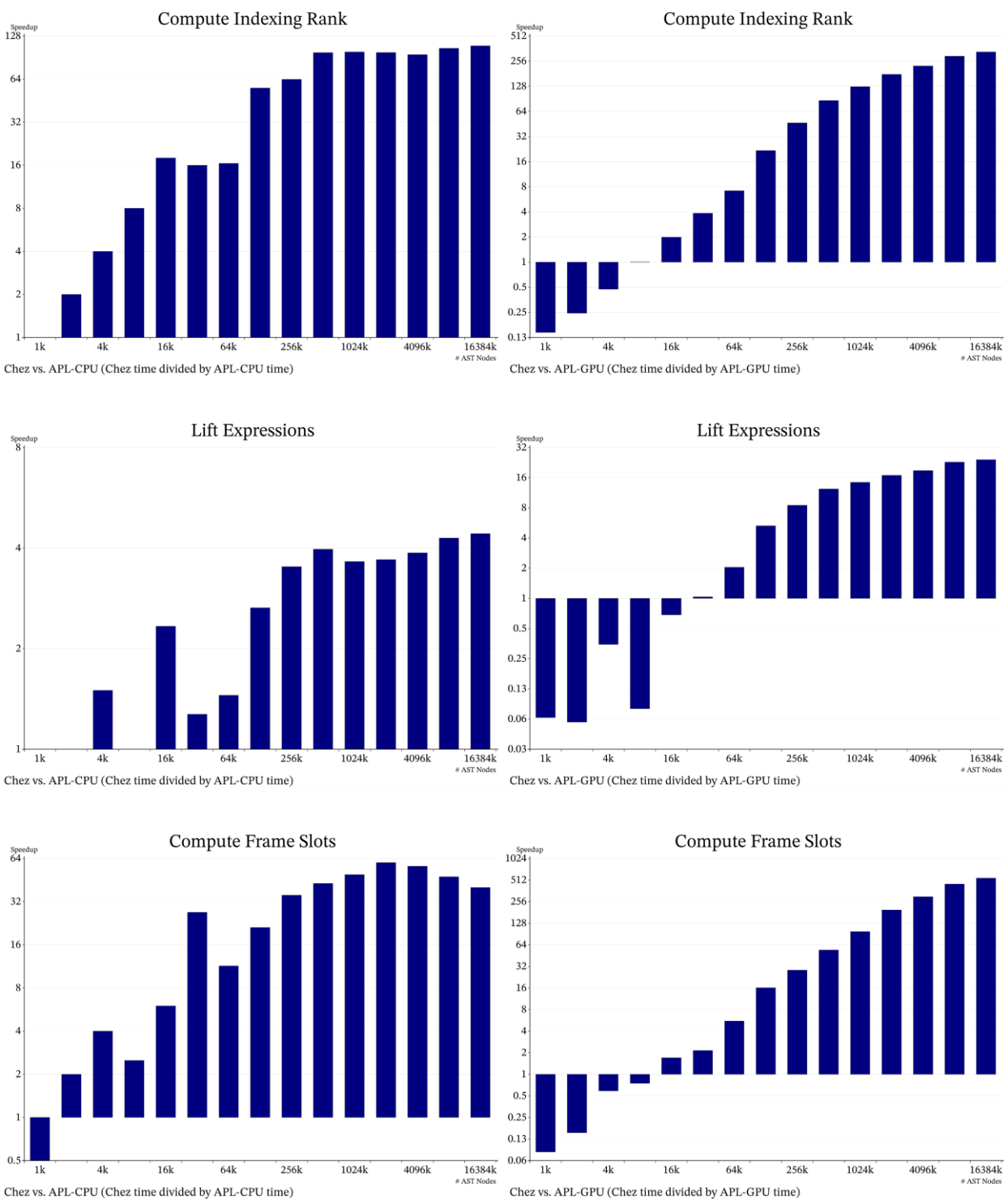


Figure 9. Speedups vs. Chez Nanopass for CPU (Left) and GPU (Right) platforms, continued.

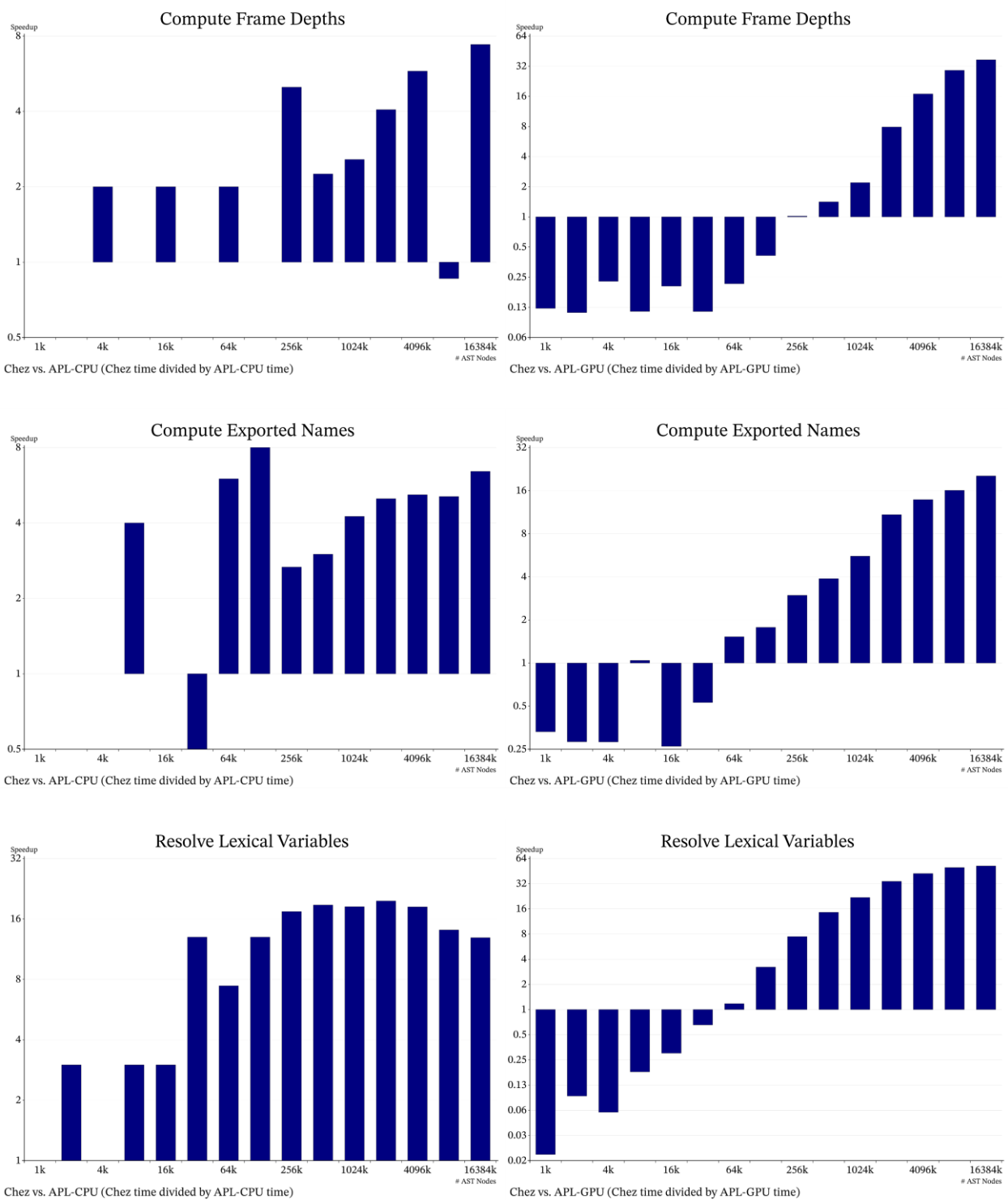
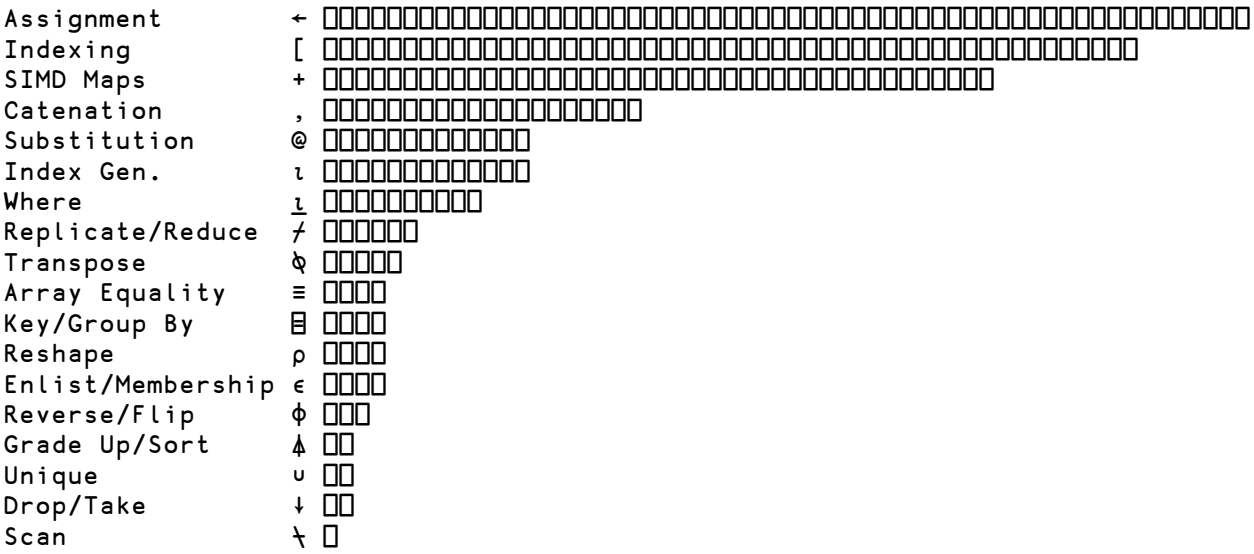


Figure 10. Speedups vs. Chez Nanopass for CPU (Left) and GPU (Right) platforms, continued.

5. DISCUSSION

5.1. Distribution of Array Primitive Usage

The following histogram illustrates the classes and types of operations used by the compiler based on the number of times they appear in the source code:



Indexing is by far the most common main operation when combined with SIMD scalar primitive operations. These operations show the relative importance of each type of operation in the code, as well as indicate the specific features required by any runtime library that hopes to implement these features.

5.2. Tree Transformation Idioms

The techniques in Section 3 break down into a few different classes of operations over trees. These operations represent idiomatic tree manipulations worth considering as specific strategies and approaches to data parallel tree mangling. Pointer vectors represent the core data representation over

which these operations work. They fall broadly into the categories of traversal, edge mutation, and node mutation. Traversals walk the tree to obtain information, sometimes storing information back into the tree, without modifying the structure of the tree. Edge Mutation doesn't change the number of nodes in the tree but changes the edges so that the nodes move around within the tree. Node Mutations change the number of nodes in the AST through addition or deletion and usually also involve some sort of edge mutation.

5.2.1. Traversal

Most compiler passes require traversal over the tree. In a traditional compiler design, these traversals exhibit a high degree of uniformity and almost always use a depth-first traversal pattern beginning at the root node of the tree. This pattern heavily dominates the space as the de facto standard with OOP Visitor patterns, as well as the vast majority of "tree traversal" patterns in languages like Scheme or Haskell. The Nanopass framework, which deeply informs this work, encodes depth first traversal through numerous syntactic aids to automatically traverse trees without explicit recursive calls. In Nanopass, the source code intermingles the modification of the tree within and under this core depth first traversal pattern.

Contrasting this, Co-dfns compiler passes generally contain a distinct traversal phase to identify nodes; modification and structural changes to these nodes occur after the traversal phase. This separates the traversal pattern from the structural modifications. Sometimes, selecting the right nodes represents the core logic and much of the source code within a given pass.

Furthermore, Co-dfns does not rely on a single unified traversal pattern such as depth first traversal. Rather, it uses three broad strategies, each of which selects nodes based on progressively more complex criteria and arrangement: they are Each, Walk, and Key.

The Each pattern suffices for most traversals. This pattern mirrors the Each (¨) operator in APL, which captures the concept of applying a given function to each element in its argument. Other languages call this Map. Scalar function primitives of APL implicitly embed a recursive Each semantics into their traversal logic. The Each pattern represents the simplest logic because it does not account for edge information. Instead, an Each traversal works if the information desired can be found by examining each node in the tree independent of every other node. In such a case, the Each pattern is usually the tidiest and tends to be the most efficient. It does not require any traversal of the edge data in the tree and is agnostic to any specific locality optimizations on the order of the nodes in the pointer vector. Its access pattern is memory optimal even in the presence of significant node order permutations in the parent vector. It is perhaps surprising the number of compiler passes amenable to the Each traversal pattern. In the description of specific compiler passes in Section 3, the Each patterns are those that require only the application of trivial search or scalar primitive operations over the field data, without requiring the "walking pattern" to walk the parent vector.

In cases where a pass requires the parent-child information in the tree, then there needs to be a way to traverse the tree according to its edge data. This often relies more on parent information than sibling, which makes the parent vector so useful storing the edges of the tree. Recall the following walking idiom to traverse up the tree described by a parent vector:

```
I@{...}⍟≡⍨p
```

This idiom enables the collection of information on each node in the tree utilizing its parent information. As discussed in detail in prior sections, this idiom emphasizes traversing from the bottom upwards, which improves the predictability of available parallelism. Conceivably, by tweaking the ordering of the parent vector, locality for certain access patterns could be improved based on hardware specific knowledge leading to improvements in overall memory throughput, but this was not explored in the benchmarking efforts of the previous section. While many traditional problem formulations bias the description of a solution towards top-down traversal, the treatment above demonstrates that many such traversals also admit a bottom up solution, thus making them amenable to the walking idiom. Additionally, while the walking idiom works as is for many cases, some access patterns require modification of the idiom in various ways, as seen in the lexical resolution pass that uses a more sophisticated walking pattern to filter out unnecessary elements due to the cost of each step.

Sometimes a transformation requires working directly with the children rather than traversing bottom up. In such cases, the Key pattern is useful. In Section 3, the Counting Indexing Rank pass includes an alternative formulation to the bottom-up traversal using the Key pattern. In this pattern, the appropriate nodes are identified and then grouped by using the Key (⌸) operator to enable specific operations over the children at the "cardinality" of the parents. Note that this approach involves a greater potential computational effort depending on the shape of the inputs and the operands, and it is therefore not usually worth the additional cost. Moreover, without careful consideration to the cost

of inner computations, it is easy to specify an operand with a critical path too great for practical use on parallel hardware.

When traversing, the Each pattern, when written correctly, generally has a constant time critical path or one that is logarithmic in the size of the tree, but with very low constant factors, since the logarithmic elements generally derive from a final where (ꞟ) primitive. The Key pattern can be logarithmic if written correctly, but it will usually involve some level of sorting. With an appropriate operand, the Key pattern can be a low overhead operation. Care should be used with the Walking idiom to ensure that each iteration has the smallest possible data and/or critical path as the overall critical path of the Walking idiom derives from the fixed point over each iteration, making the critical path of the iteration critical to good performance. Unnecessarily expensive iterations can result in excessive overheads, even if the critical path is normally bounded to the depth of the tree.

To mitigate this issue, when working with walking idioms, note that it is not necessary to walk only along the parent vector. Often, the real path to walk can be formulated as something with a smaller number of steps, such as in passes that leveraged the referent field above. Pre-computing a walking path in the form of a separate pointer vector, to use in future computations instead of using the parent vector for traversing the tree, allows you to avoid intermediate steps and enables a lower effective critical path. This “skipping” concept appears elsewhere in the use of skips in skip lists and other tree traversal techniques, sometimes called ropes (Goldfarb, Jo, and Kulkarni 2013). In the compiler above, the referent field serves this purpose, and its use helps quickly identify both the next lexical scope as well as the nearest function. These paths amount to a new tree that shares nodes but

uses different edges as the main tree described by the parent vector. They serve not only to improve walking speed, but also help to color nodes into classes, thus serving useful purposes for the Each and Key patterns, as well. The creation of custom walking paths should be considered a fundamental strategy for improving code efficiency and clarity.

Finally, the canonical structure of the tree sometimes contains data that simply cannot be clearly accessed in an efficient manner through the access patterns exposed via the tree structure. In these cases, it is often better to extract this information out into a form that suits the anticipated access patterns. As a classic example, the environment table of bindings created during lexical resolution enables a rapid search of the bindings within the tree on each step of lexical resolution without requiring a walk through the tree to find each binding each time. While this technique may seem obvious, reducing the cost to create these structures enables their more liberal use.

5.2.2. Edge Mutation

Edge mutation is the process of changing the order and shape rather than the cardinality of the tree. Generally, there are two ways to do this, by changing the parents of various nodes or by reordering the sibling relationships.

The required mutations differ little between the traditional compiler formulation and those described here; the differences come from structure, representation, and order of operations. In a traditional compiler, a pass tracks the nodes that will be new parents of a given set of nodes and then traverses through the tree to identify the new children, associating the new parents with the appropriate new children by updating whatever field or structure records the children. The traditional

compiler pass mingles the process of traversal, collection, and update within the source. In the Co-dfns compiler, the traversal generally occurs first to identify nodes to update, followed by collecting the nodes together into their appropriate groups. After identifying the appropriate parent and child nodes, the child nodes receive new parents through a bulk update to their parent vector positions. This separation enables for the traversal to occur without respect to collection, for the collection to work over the whole tree, and for the update/mutation of the parent elements to occur in bulk as the final step.

In the elucidated design above, partially ordering the parent vector simplifies construction over an explicit sibling pointer vector. Reordering the siblings involves a little bit more than just updating a pointer vector, such as is the case with changing parents. The permutation of the siblings follows a similar pattern to that of updating the parents: identify the appropriate nodes through traversal and determine the appropriate permutation, then permute the nodes. However, reordering the nodes potentially invalidates the parent vector, requiring correction to the pointers in the parent vector and other pointer vectors to align with the permutation. This requires additional indexing to correct pointer vectors to point to the right places after permutation.

Edge mutation generally falls into one of the above classes, that is, either permutation of the nodes or updating pointer vectors that describe the edges of the tree, with the proviso that all pointer vectors receive updated values after permutation.

5.2.3. Node Mutation

Node mutation changes the cardinality of a tree. When adding or deleting nodes from a tree, one almost always modifies the edges of the tree as well. Nonetheless, the node mutation deserves consideration as a separate topic.

Adding nodes to the tree is the canonical operation; as demonstrated below, node deletion represents a special case of node addition. Node can be introduced in the middle of the tree, that is, by inserting them into the middle or beginning of the parent vector, or they can be added at the end of the vector. This distinction arises because of the difference in fundamental performance between appending to the tail of a vector and inserting new elements somewhere in the middle or beginning: appending to the tail does not change any other node ids, and therefore requires no recomputation of pointers, while adding into the middle or beginning of a vector does.

All other things equal, code should always use tail catenation when possible, because of the large efficiency benefit. When the introduced nodes are all to the right of or unordered with respect to existing nodes, tail catenation works well. If a transformation requires multiple additions to the left instead of the right, then reversing the pointer vector may still enable tail catenation within this set of additions, with only the single reversal cost. Moreover, when adding nodes, adding as a single bulk addition almost always nets better performance over smaller individual additions.

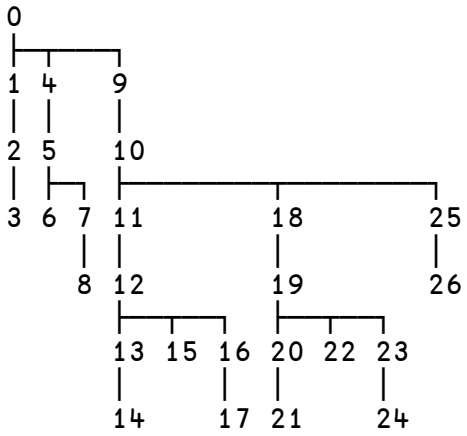
When other factors negate the advantages of tail catenation (such as preservation of a specific ordering for use in a later pass, as with the Expression Wrapping pass), then one must splice nodes into the middle of the parent vector. APL provides convenient expand and replicate operations to

efficiently create space for doing this. Doing so with a replicate or expand also eases pointer recomputation as it is straightforward to predict the location of new nodes.

A careful reading of Section 3 will reveal that no nodes were deleted during the compiler operations. One might therefore presume that the above technical demonstration does not adequately address the issue of node deletion. Nodes may be removed in two ways, they can be marked as dead and then ignored, while leaving them in the vector, potentially reusing the space at a later time, or the same replication techniques used to splice nodes into the tree may be used to remove them. Replicating a node 0 times removes it from the tree, and the same techniques for pointer recomputation used for splicing work for deletion. Thus, while the compiler does not explicitly address node deletion, the same techniques used for splicing work for deletion.

However, it is worth considering an alternative pointer recomputation idiom for node deletion as a cute demonstration of the use of Interval Index. It represents a useful general technique and has wider application. This technique could be termed the “garbage collection” idiom, getting at the fact that memory management in the Co-dfns style does not require garbage collection, though tree manipulation represents a classical case for the benefits of garbage collection in functional programming languages.

Consider the following AST with node ids:



Consider the task of deleting nodes 4, 12, and their children. The first step selects those nodes and identifies them as the set of nodes to delete, using the selection and traversal idioms:

```
      ⍳M←4 12∊¨p I@{~⍵∊4 12}⍤≡⍳≠p
4 5 6 7 8 12 13 14 15 16 17
```

The mask `M` helps to create the new parent vector with the appropriate nodes removed:

```
      p
0 0 1 2 0 4 5 5 7 0 9 10 11 12 13 12 12 16 10 18 19 20 19 19 23 10
25
      (~M)⌿p
0 0 1 2 0 9 10 10 18 19 20 19 19 23 10 25
```

While this removes the elements from `p`, the values of `p` still point to their old locations. The garbage collection idiom shifts the pointers to the correct, new locations efficiently and succinctly:

```
      (⍳M)(⊢-1+⍳)(~M)⌿p
0 0 1 2 0 4 5 5 7 8 9 8 8 12 5 14
```

The idiomatic train `⊢-1+⍳` takes as its left argument the deleted nodes and as its right argument the compacted pointer vector to correct. Node deletion implies that the pointer values can only ever shrink, and thus the fundamental task is to determine the amount by which to reduce each pointer.

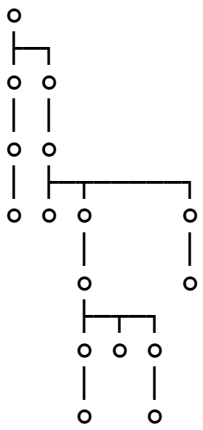
The shift amount for any pointer corresponds to the number of nodes that were deleted prior to the position pointed to by that pointer. This is precisely one more than the value computed by the interval index function ⍳:

```
        (⍳M)(1+⍳)(~M)⌿p
0 0 0 0 0 5 5 5 11 11 11 11 11 11 5 11
```

For any pointer, this GC idiom recomputes the pointers on a recently compacted pointer vector. Because this process is explicit, it permits the design of the compiler to determine the appropriate time for compaction, giving control over the most advantageous points to compact the tree, potentially avoiding unnecessary compaction overheads. After compaction p looks like this:

```
0 0 1 2 0 4 5 5 7 8  9 8  8 12 5 14
```

And represents the following tree:



5.3. Benefits of This Style of Tree Transformation

The Co-dfns approach to compiler construction provides some benefits beyond the performance of execution, not the least of which is a fundamentally simplified model for reasoning about asymptotic complexity. Since function composition from a core set of well-studied primitives comprises nearly

the whole of the design, where each primitive has well known asymptotic behaviors, calculating computational complexity follows simple algebraic rules. The sole complication in this is the few locations where the fixpoint idiom requires proof of termination.

Of course, the sheer compactness and directness of the code cannot be ignored. The source code demonstrated here fits into 17 lines of direct code what takes approximately 1000 lines in Nanopass. Note that it does this using the APL language directly, without additional abstractive overheads, whereas the Nanopass design is a domain-specific language targeting compiler construction from the outset. The APL code demonstrates significant simplification and reduction of the abstractive and syntactic overheads compared to Nanopass. That is, the methods here are directly accessible to anyone with familiarity in APL or Iverson style array languages, and the code would require no additional documentation in terms of intermediate abstractions or supporting frameworks, compared to using Nanopass or the like, which utilizes a number of highly specialized techniques in order to provide its syntactic convenience, and obscures significant portions of the underlying implementation behaviors behind library boundaries that need to be unpacked in order to fully understand the performance model. Even compared to other direct implementations, such as might be seen in Haskell or the like, the APL solution exhibits significantly better economy and requires significantly fewer fundamental concepts in order to understand its operation. The current literature lacks adequate controlled trials to suggest whether this approach fundamentally alters the readability, usability, programmability, or reliability of the code, but at least in terms of sheer complexity on this one example, the Co-dfns approach is fundamentally simpler.

Moreover, when written in this style, implementing introspective and interactive debuggers requires less effort, and current somewhat direct and naïve debugging facilities available within Dyalog APL are enough to introspect the behavior interactively not only between passes, but also within passes at several levels of granularity. Functional programming approaches still do not yet have this level of introspective ease when applied to the domain of compilers and trees, particularly when examined in terms of return on effort invested. As an example, most of the tree diagrams within this work were generated within an interactive debugging session by running the compiler on the desired source at the appropriate time. The rendering of the trees was accomplished with the code seen in the appendix. This required only the 8 lines of rendering code found in the appendix and the standard debugger. The directness of the approach means that all the standard features available within a typical APL system apply equally well to trees, including binary and textual serialization with or without compression, network/system portability, arbitrary suspend and recover, etc. without any special code to take advantage of these benefits. The compactness and directness of the compiler expressions greatly enhances the ease with which these features may be used.

Section 4 demonstrates the scaling of the compiler over a wide range of data. Note the scaling of the compiler at multiple granularities. The design of the compiler makes it inherently agnostic to module count. That is, it compiles any number of modules in the same way that it compiles a single module. There is no distinction between the compiler and a parallel build system orchestrating multiple instances of a compiler. It is also agnostic to different sizes of modules compiled together, since the compiler only sees the modules as a single tree, thus building in a sort of load balancing effect

that enables small and large modules to be compiled together and the work distributed evening. This allows the compiler to perform well on small modules while at the same time scaling up to any number of differently sized modules with more predictable performance. It accomplishes this without the need for any explicit logic to do so and without any need for additional tooling or infrastructure support from a parallel build system. This can further simplify the compiler architecture by permitting parallelism and inter-module optimization without the orchestration overheads, since the same compiler instance can work on the modules together and optimize them together in a single, parallel instance, without the extra synchronization overheads involved in a design that relies on parallel build systems for scaling.

While inspired by the Nanopass methodology, unlike Nanopass, the compiler contains no strict abstractive barriers between each pass. Thus, an optimizing compiler may more easily do cross-transformation optimizations than might be possible if hard abstractive barriers were in place. Such a compiler may be able to provide the theoretical benefits of a mono-pass design without the associated complexities. Some such optimizations could include lifting memory and sanity checks to a pre-execution phase, thus encouraging further opportunities for parallel kernel fusion and cache optimizations.

The compiler as written demonstrates significantly better hardware platform independence, since the same source code, despite being written at a high-level and without particular attention to the hardware requirements, can execute with efficiency both on small-width vector machines such as general purpose CPUs, as well as wide-vector lane machines like GPUs, each of which also has

different memory and execution architectures. Note, this efficiency comes even when executed by unsophisticated interpreters or compilers. That is, the methods described here achieve high-performance without sophisticated compiler analysis or other optimizations and relatively small runtimes (Whitney 2009).

Arguably, the compiler style as written has a sort of directness not visible in other approaches. Expressions such as `p[i]←p[x]` can be read as, "the parents of nodes i are the parents of nodes x." Whether a programmer values such "lyricalness" (Perlis 1977) is probably a matter of style, but the approach exposes the specific structural modifications and the traversal patterns more directly than the embedded edge modifications of a functional recursive approach. The simplicity of this directness may have positive ramifications on the formal verification of such compilers, since the operations are fundamentally simpler to formalize and reason about than equally performant and parallel implementations written in other languages. Formal understanding of parallel constructs of the type used in this compiler are more accessible and well established, whereas the author is not aware of significant research on the formal verification of recursively defined parallel operations that fundamentally modify irregular tree structures.

Directness means little abstraction, infrastructure, scaffolding, or toolchain support to express the solution. While the notation itself provides significant advantages, the semantics of the language itself are extremely simple, and the model accessible to a wide range of languages. Thus, the machine performance advantages are widely accessible. There is no need for specialized libraries, type systems, compilers, or other infrastructure tools in order to begin seeing benefits. Of course, more sophisticated

runtime and compiler optimizations will improve the performance capabilities, and better notations for the programming model (namely, APL) greatly facilitate the use of these techniques, but in principle, most platforms and systems can allocate contiguous memory and efficiently iterate over that memory. Nonetheless, the author does believe that significant syntactic and linguistic support for this model greatly aids in its use, and languages that naturally support this model of programming at their core layer are more likely to scale—particularly as it relates to human usability—with this approach than languages that emphasize other models of computation.

5.4. Limitations of the Approach

The greatest limitation of this construction is unfamiliarity. The approach described in Section 3 bears little resemblance to the prevailing programming practice for writing compilers. This presents an undeniable barrier to initial entry and application. It is not clear whether the construction described can be adequately targeted through compilation or transformation of more traditional methods. Nonetheless, this unfamiliarity stems largely from within the domain. The use of Iverson style array programming outside of tree transformations has a long history and has a strong pedagogical practice. Thus, the wider application of data parallel programming in everyday use will only improve the ease of learning to apply such programming to compilers and other tree transformations.

Another current limitation is the dearth of statically typed features in current APL language implementations, and it remains an open question whether explicit static typing in the code would be a fundamental improvement. As noted previously, existing typing systems for APL fail to meet adequate standards of usability for the typical APL programmer (Slepak 2014; Gibbons 2017).

Finally, no matter how fast or potentially scalable any of the tree transformations given here are, they are limited by the total impact on the broader application design. In systems that spend relatively little time on tree transformations, but significant amounts of time on parsing or analysis passes, then the direct impact of the tree transformation algorithms may be limited. On the other hand, in cases where a significant amount of work is spent on tree transformations, with very little or easily amortized costs in terms of parsing or other operations, then these operations obviously contribute more heavily. Finally, the availability of high-performance tree transformations on the GPU contributes an indirect performance gain, in that they can enable data to remain longer on the GPU and can serve to glue together more expensive algorithms that require different data formats or layouts without needing to process this information on the CPU. This avoids the data transfer overheads usually involved in GPU computations, which can often create a hard limitation on the potential available parallel speedup for applications in which significant time is spent on transferring data to and from the GPU.

5.5. The Readability and Usability of APL

It cannot be ignored or escaped that this work relies heavily on APL as its programming model, which has a reputation for being hard to read and "write only." Perhaps the predominate question when this work is examined among the wider programming community centers around APL as a suitable language for "real work." The broader programming community has questioned the real usability of writing a compiler that appears to be so dense and so concise, particularly with the high level of

symbols used. The apparently unorthodox approach to encoding and working with the trees also draws significant questions as to the “readability” and maintainability of the code in the long term.

Unsatisfactorily, the author has not conducted sufficiently conclusive usability studies on the topic. Without asserting unequivocally, the process of developing the compiler has born some points of discussion on the matter. Concerning the use of symbols, the models used and the approach taken could just as easily have been written without recourse to the symbols. The syntax is simple enough that the same basic algorithms could be encoded into a more verbose style. It is the author’s personal experience that the difficulty in learning or working with this model of programming for those who are not familiar with it lies not in the symbols (which are, in many ways, a boon and assistance) but rather with the alien thinking involved with the data model and the primitives used in a way that is not familiar to the reader. To give an illustration of this, consider the Wrap Expressions code:

```
i←(⍳(~t∈3 4)∧t[p]=3),{ω≠⍨2|⍳≠ω}⍳t[p]=4 ⋄ p t k n r≠⍨←⊂m←2@i⊢1⍴⍨≠p
p r i I⍨←⊂j←(+⍀m)-1 ⋄ n←j I@(0≤⊢)n ⋄ p[i]←j←i-1
k[j]←-(k[r[j]]=0)∨0@({⊃⍵⍵}⌸p[j])⊢t[j]=1 ⋄ t[j]←2
```

Now consider the same code transliterated into Scheme-like notation:

```
(define i
  (catenate
    (where
      (and
        (not (members-of t `(3 4)))
        (pequal? 3 (indexing t p))))
    ((lambda (w)
       (replicate
         (modulo
           (index-gen (tally w))
           2)
         w))
     (where (pequal? 4 (indexing t p))))))
```

```
(define m ((at 2 i) (reshape (tally p) 1)))

(define-values (p t k n r)
  (let ([m^ (enclose m)])
    (apply values (map (lambda (x) (replicate m^ x)) p t k n r))))

(define j (pminus (scan-first + m) 1)

(define-values (p r i)
  (let ([j^ (enclose j)])
    (apply values (map (lambda (x) (indexing j^ x)) p r i))))

(define n
  ((at indexing (lambda (x) (pless-equal? 0 x)))
   j n))

(define j (pminus i 1))

(array-set! p i j)

(array-set! k j
  (negate
    (or
      (pequal? 0
        (indexing k
          (indexing r j)))
      ((at 0 ((key (lambda (x) (disclose (reverse x))))
              (indexing p j)))
       (pequal 1 (indexing t j))))))

(array-set! t j 2)
```

Let the reader judge whether removing the symbols fundamentally enhances readability and clarity.

Putting aside the question of symbols, there is still the question of general accessibility, and the method's viability as a primary mode of compiler authorship. In addition to the points about debuggability and introspection above, there is one anecdote worth conveying.

An older version of this compiler written in a less clear and more complicated style (by anyone's estimation) was presented at Iverson College.[2] There, two language implementors who were

[2] http://www.iversoncollege.com

not compiler writers, but who programmed in array languages (though they were not strictly APL programmers), wished to examine the compiler with the explicit requirement that they received no specific assistance from the author with regards to interpreting or understanding its make-up, architecture, intent, or the like. They examined the code without comments in the same terse style given here. Within a few hours, the two implementors were able to divine not only the structure of the compiler, but also the semantics implied by a number of the compiler passes, and began asking important questions about whether the semantics implied by the compiler passes were indeed the ones that APL followed. In other words, they were able to divine and compare the semantics implied by the tree transformations encoded in this style of compiler construction within a few hours of having first seen the code without any documentation or support and only an interactive session with which to experiment on the compiler and test it with various inputs.

The compiler architecture at that time used a style significantly more complex, with less clarity than the current model. While this certainly falls far short from anything approaching proof of the readability and maintainability of this approach of tree transformation, it at least demonstrates that other array programmers (not APL programmers) who are familiar with the programming model, but not with the techniques of compiler construction were able to understand the tree operations performed by the compiler rather quickly without any explicit documentation. This might be compared to the requirements and process of learning a standard compiler architecture, its associated documentation requirements, and any specialized framework or language features before understanding the behavior and semantics implied by various compiler passes.

6. FUTURE WORK

Many avenues currently present themselves as extensions to the work presented here, a few of which are discussed below.

An obvious first step is the formal, mechanical asymptotic analysis of the compiler. The simplistic nature of the control flow and the reliability of the asymptotic behavior of the APL primitives strongly suggests that the design of the compiler is such that an automatic analysis of the computational complexity of the code is not only possible, but extremely accessible. Given that the compiler is written in a style designed to be mimicked by other writers of APL, the availability of such a code analysis pass in the compiler itself would greatly enhance the value of writing such "idiomatic APL."

The compiler passes and their implementations in this work have been tremendously simplistic, with no additional optimization of the passes beyond the trivial implementation niceties of the core algorithms as they appear in the text. For both the GPU and the CPU, there are obvious opportunities to improve the data locality of individual passes and to optimize the ordering and structure of the objects in such a way as to facilitate better read-write performance. This deserves careful exploration.

The emphasis in this text has been that of tree transformations as found in compilers, as this is a well-known, and well contained set of related and synergistic algorithms. However, the generality of this work means that it should readily apply to other work, such as garbage collection, document generation, DOM processing, and other such operations. Verification of this appearance would

enhance the practical evidence of the generality of these methods beyond the current evidence in this text.

One optimization that would also appear to have some benefits is the ability to separate the allocation phases from the kernel executions during compile time. The benefit of this is the potential to pipeline certain allocation operations, as well as certain data transfer operations. Additionally, with better analysis of the read and write requirements (perhaps as a part of the asymptotic analysis mentioned above), more efficient use of memory could be found by separating and pre-allocating the memory and working space before execution. This also presents interesting opportunities for fusion.

In addition to this pre-allocation of memory, additional optimizations exist to reduce the amount of memory used during execution on the GPU. The GPU implementation presented in this text is extremely naïve, and as such, uses more memory than it needs to use, despite having the appropriate asymptotic memory usage. Compile-time optimizations and more efficient runtime implementations would both contribute to improving the overall memory requirements.

Currently the generated code utilizes some heavyweight runtime libraries in order to facilitate execution. By utilizing more specialized custom kernel generation it is possible that kernel usage as well as runtime overheads may be reduced. This would require more sophisticated runtime code generation (assembly) as well as more sophisticated analysis in the compiler itself. Versions of this analysis have been prototyped (Hsu 2014, 2015) but these were removed for the purposes of this text in favor of the simpler treatment, and because they did not measurably contribute to the different types of compiler passes and tree operations demonstrated here. It was also a goal in this text to remain

as simplistic in operation as possible, and to demonstrate a baseline performance for these techniques, rather than to employ sophisticated and specialized optimizations. Reintroducing such analysis and code generation is likely to improve performance.

The emphasis of this text has centered on the compilation of the Co-dfns syntax as an untyped language. However, models of a typed form of dfns exist, and it would be interesting to utilize the techniques discussed here to accelerate the typing and compilation of statically typed languages as well.

A side-effect of the simplicity of the compiler is that it is also an accessible target to formal verification. The use of APL as the implementation language creates an appealing formal verification target that is likely to be more accessible to verification than other parallel implementations of compiler passes in other models.

While the APL given in this text is platform agnostic, there are cases in which one or another approach to implementation will result in higher performance on a GPU vs. a CPU. It would be interesting to explore a "rewriting" pass in the compiler that automatically rewrote parts of the compiler for optimization based on the execution platform at an APL level. This would require reasonable levels of idiom recognition to be built into the compiler. Improved idiom recognition and fusion passes will likely improve the compiler performance across the board, though it is unclear how much this would improve the overall compilation performance tested here.

Right now, the compiler does not have any support for the execute primitive, which enabled for the runtime evaluation of expressions. This is a classic challenge for compilers, and an interesting

question is whether a parallel interpreter on the GPU can possibly exist in any meaningful way, or whether a hybrid CPU-GPU approach could be designed that would work as well. This would enable the compiler to support the execute/eval primitives.

Finally, this work focuses expressly on the compilation aspect of the design, which are the tree transformation after parsing and before code generation. However, parallel code generation and parallel parsing are both explored ideas, and it would be interesting to combine these together with the parallel tree transformations presented here to examine a “total compiler” benchmark that includes parsing and code generation within it.

7. RELATED WORK

The J programming language (Hui 2007) was the first practical, general-purpose programming language to introduce the Key operator as a primitive operator with the presumption of its general usefulness.

Robert Bernecky (2015) argues that the increased programmability and desirable human factors of APL-style array programming can be implemented with competitive performance relative to more traditional techniques. This suggests that programs written in this highly abstract style may not suffer the performance gap traditionally assumed to go with their desirable features. This work is bolstered by previous work on the high-performance implementation of array-oriented languages (Ching 1990; Ching and Katz 1994; Ching, Carini, and Ju 1993; Ju and Ching 1991; Ju, Ching, and Wu 1991; Bernecky 1999; Schwarz 1991).

Fritz Henglein demonstrated a class of operations, called discriminators, of which the Key operator is a member (2013), 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 (Prabhu et al. 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 path matrix and parent vectors described in this work, though they have a different formulation and an approach that emphasizes the analysis pass, as opposed to the structural modifications emphasized in this work.

Mendez-Lojo, Burtscher, and Pingali implemented a GPU version of Inclusion-based Points-to Analysis (2012) that also focuses on adapting data structures and algorithms to efficiently execute on the GPU. 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 (2003). It also uses a matrix format to represent the AST. However, traditional APL did not have nested function definitions, and thus the APEX compiler does not have any specific approaches to dealing with function lifting, lexical scoping, or the like, and did not use a vectorized, parallel approach to working with the AST during the primary tree transformation efforts.

Timothy Budd implemented a compiler (1984; 2012) for APL which targeted vector processors as well as C. Budd provided thoughts and some ideas on how the compiler might be implemented in parallel as well.

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 (1984).

There has also been significant work on irregular tree traversals, as opposed to transformations. The work of Goldfarb, Jo, and Kulkarni (2013) demonstrates a method of building "autoropes" for jumping between nodes efficiently during traversal to avoid extra loads during tree traversals. These methods, based on prior "rope" based methods, is very similar in spirit to the use of

pointer vectors in this thesis, but defer in encoding, approach, and overall design because of their emphasis on retaining the recursive algorithmic structure, as well as the continued emphasis on top-down traversals. They also proposed a dynamical schedule that allowed for improved traversal locality based on prior traversal behaviors (Jo, Goldfarb, and Kulkarni 2013). Feng Zhang et al. have addressed the issue of tree traversals through a pre-processing that allows for the reordering of operations to ensure efficient access on the GPU to various nodes in tree traversals (Zhang et al. 2016). Liu, Hegde, and Kulkarni have also proposed a hybrid CPU-GPU scheduling approach that allows for a multi-stage approach where the CPU is used to perform scheduling operations before this information is sent back to the GPU to enable more fine-tuned algorithm execution (2016). Ren et al. (2012) divide the work of irregular tree traversals using a specialized SIMD interpreter which, much like this work, dedicates single threads a priori to specific traversals. However, they acknowledge the overhead involved with their approach, despite seeing improvements in overall performance.

8. CONCLUSION

Implementing parallel compilers and other parallel tree transformations has proven challenging and fraught with exceptional and dangerous corner cases. Due to irregular shapes of the trees, the seemingly hard to predict branching factors hiding available parallelism, and the heavyweight costs to thread creation relative to the available work, tree transformations have proved elusive to parallelize in a general, scalable, platform independent manner. The usual approaches focus almost entirely on traversal, as opposed to transformation, and efforts to parallelize compilers have resulted in the discovery of deep, systemic limitations within a compiler design or an architecture's ability to effectively utilize available parallelism both in terms of programmer usability or speedups on the machine. Prevailing opinion often concludes that such parallelization efforts are not worth the likely results because of the relatively coarse-grained levels of parallelism and scalability expected. In the end, most systems tend to use single-threaded tree transformations orchestrated through parallel build systems of one form or another instead of attempting to apply a universal model of parallelization to the core compiler process.

This work suggests a different conclusion. The compiler as described in this work strongly suggests that the difficulty of parallelizing tree transformations at scale and in a general way rests not upon any inherent difficulty in the nature of tree transformations, but in the foundational presuppositions of the appropriate data and operational abstractions, and that a complete reconsideration of these foundational abstractions can not only thoroughly address the challenges of parallel tree transformations, but do so in a way that has significant additional benefits. Rather than

invent ever more sophisticated programming models, systems, and architectures designed to ameliorate the traditional problems with parallel tree transformations, this work proposes a nearly opposite approach, to remove almost all the sophistication and levels of abstraction usually present in such systems and to studiously avoid the introduction of any novel paradigms or execution models or support.

Furthermore, by effectively moving memory-bound tree transformation algorithms to the GPU, more compute-bound algorithms such as optimization analysis passes that have existing high-performance GPU implementations can avoid the need to transfer information back and forth from the GPU to the CPU main memory, reducing the amount of time spent on expensive data transfers.

The compiler described above begins by eschewing the prevailing practices of tree transformations down to the fundamental data abstractions and representations. Rather than begin the work of compiler construction with traditional tree abstractions and traditional data models, it begins by selecting a programming language and abstraction suitable for fine-grained, SIMD parallel execution, APL. This model is almost the exemplar of “not new” in terms of its core programming paradigm, but it nonetheless receives little attention in this space. By adhering to a strictly idiomatic approach to composing and writing APL this naturally results in a parallel program by construction. The compiler is built on this alternative data model and reorients the construction process to take a parallelism first approach, rather than attempting to layer parallelism on top of more traditional algorithmic expressions.

The resulting compiler exhibits remarkable characteristics. At the source code level, it is fundamentally simpler, requiring only 17 lines of code compared to the approximately 1000 lines of code used in a more traditional, but state of the art, compiler framework such as Nanopass. This code is fundamentally simpler not only in line count, but also token, depth, structure, definition and use distance, variables, library requirements, abstractive overhead, and most other metrics. The source is significantly easier to compile or execute with high performance, as demonstrated by the high-performance results achieved both with interpreted and compiled implementations, neither of which have "sophisticated" compilation or analysis strategies. These 17 lines outperform the Nanopass equivalent by up to 200 times on average across the entire compiler on the GPU, and up to 33 times on average when executing on the CPU on an interpreted environment (vs. a compiled environment for the Nanopass compiler). The average compiler speedups are but a part of the story. Not only is the average performance better, but on every individual compiler pass, the performance exceeds the traditional approach most of the time with few exceptions. The traditional approach can compete with the performance of the CPU implementation on only a few passes at small data sizes and requires significant compiler and runtime support in order to do so. It is also surprising that the GPU performance regularly outperforms the traditional approach even at very small data sizes, whereas traditionally GPU acceleration requires significant data sizes to begin seeing improvements.

A significant contributing factor to the performance difference between the traditional approach and the Co-dfns approach described here is the sheer magnitude of difference in memory usage in the chosen data models, as well as that data's locality. The traditional method uses

significantly more memory, and it does so in a fragmented way, leading to less memory locality and poor memory access patterns relative to the Co-dfns approach, in which the memory access patterns are not only favorable, but can be reasoned about directly. This "reasonability" also scales to the execution and asymptotic complexity of the code, where the calculation of computational complexity, memory usage, critical paths, and other theoretical performance metrics are all extremely straightforward, because no complex control flow paths exist in the code, leading to the majority of the analysis coming from predefined primitive functions applied in sequence. While this work does not attempt to prove that the complexity is machine computable, the simplicity of the analysis strongly suggests that this is a computable problem readily accessible as future work.

Thus, the compiler is shorter, faster, leaner, simpler, easier to analyze, easier to compile, exhibits better performance portability, and does all of this while utilizing extremely simple and straightforward semantics and programming models that require no sophisticated programming language infrastructure support, and could be in theory implemented straightforwardly at a basic level in almost all present day commercially utilized programming languages. One of the most novel aspects of this work is that this is achieved without the introduction of any novel programming models, paradigms, runtimes, execution engines, analyzers, or even design patterns. The programming techniques and approaches are widely recognized and used APL idioms found within the APL community, and no special primitives, syntax, or other constructs are used. The programming techniques used in the compiler would be readily recognized as common techniques to be found in any standard APL program across a wide array of domains.

The results described above and in previous sections suggests that compiler design and other tree transformation algorithms might find new life and skirt traditional performance hurdles by examining fundamentally and perhaps radically different models of foundational abstractions than the pointer-based top-down record-type representations presently in use in nearly all tree transformation applications. Moreover, it highlights a serious source of inefficiency in these record-type representations that warrants further consideration, given the magnitude of the difference in memory usage and performance seen between the array-oriented model and the record-type model. The prevailing and overwhelmingly common preference towards the use of record-types as the main source of data abstraction on systems running over parallel hardware (as all modern hardware is), particularly in times when memory bandwidth is often the key limiting factor to performance, may not be the obvious choice that it was in years past, and the author believes that this fundamental data abstraction practice deserves further study.

In summary, this work describes an alternative method for tree transformations built upon array-oriented programming paradigms and traditional APL programming that delivers a compiler that exhibits superior performance and structural simplicity compared to its peers, without requiring the introduction of any domain-specific programming techniques, frameworks, or abstractions in order to achieve these results, and without requiring sophisticated tooling support in order to achieve parallel performance across both CPU and GPU architectures. It does this without using the traditional abstractions of conditionals, if statements, explicit (inner) looping, recursion, pattern

matching, ADTs, records, threading, explicit parallelism syntax; instead, it relies almost exclusively on traditional array operations composed together using simple function composition.

APPENDIX A. APL LANGUAGE REFERENCE

Scalar

Monadic	Symbol	*Dyadic*
Conjugate	+	Plus
Negative	¯	Minus
Direction	×	Times
Reciprocal	÷	Divide
Magnitude	|	Residue
Floor	⌊	Minimum
Ceiling	⌈	Maximum
Exponential	*	Power
Natural Log	⍟	Logarithm
Pi Times	○	Circular
Factorial	!	Binomial
Not	~	
Roll	?	
	∧	And
	∨	Or
	⍲	Nand
	⍱	Nor
	<	Less
	≤	Less Or Equal
	=	Equal
	≥	Greater or Equal
	>	Greater
	≠	Not Equal

Other

	←	Assignment
	⍺	Left Argument
	⍵	Right Argument
	⋄	Statement Separator
	⍬	Empty numeric vector
	[ ]	Indexing

Mixed

Monadic	Symbol	*Dyadic*
Shape	⍴	Reshape
Ravel	,	Catenate
Table	⍪	Catenate First
Reverse	⌽	Rotate
Reverse First	⊖	Rotate First
Transpose	⍉	Transpose
Mix	↑	Take
Split	↓	Drop
Enclose	⊂	Partitioned Enclose
Nest	⊆	Partition
Enlist	∈	Membership
Disclose	⊃	Pick
	/	Replicate
	⌿	Replicate First
	\	Expand
	⍀	Expand First
	~	Without
	∩	Intersection
Unique	∪	Union
Same	⊣	Left
Same	⊢	Right
Indices	⍳	Index Of
Where	⍸	Interval Index
Grade Up	⍋	Grade Up
Grade Down	⍒	Grade Down
	?	Deal
	∈	Find
Depth	≡	Match
Tally	≢	Not Match
	⊥	Decode
	⊤	Encode
Matrix Inverse	⌹	Matrix Divide

Monadic Operators

⍨	Commute
⍤	Each
⌸	Key
/	Reduce
⌿	Reduce First
\	Scan
⍀	Scan First

Dyadic Operators

@	At
∘	Compose
∘.	Outer Product
.	Inner Product
⍣	Power Limit
⍤	Rank
⌺	Stencil

APPENDIX B. CO-DFNS SOURCE

```
⍝ A  B  E  F  G  L  M  N  O  P  V  Z
⍝ 0  1  2  3  4  5  6  7  8  9 10 11
tt←{⎕←'C' ⋄ ((d t k n)exp sym)←ω ⋄ I←{(⊂ω)[]α}
 r←I@{t[ω]≠3}⍤⍨⍤⍤p⍪2{p[ω]←α[α⍳ω]}⌿⊢∘⊂⌸d⍪p←⍳≠d                                  ⍝ PV
 p,←n[i]←(≠p)+⍳≠i←⍳(t=3)∧p≠⍳≠p ⋄ t k n r,←3 1 0(r[i])⍴⍤⍤≠i                  ⍝ LF
 p r I⍤←⊂n[i]@i⊢⍳≠p ⋄ t k(⍪@i⍤)←10 1
 i←(⍳(~t∈3 4)∧t[p]=3),{ω⌿⍤2|⍳≠ω}⍳t[p]=4 ⋄ p t k n r⌿⍤←⊂m←2@i⊢1⍴⍤⍤≠p          ⍝ WX
 p r i I⍤←⊂j←(+⍀m)-1 ⋄ n←j I@(0≤⊢)n ⋄ p[i]←j←i-1
 k[j]←-(k[r[j]]=0)∨0@({⊃⌽ω}⌸p[j])⊢t[j]=1 ⋄ t[j]←2
 p[i]←p[x←p[i←{ω⌿⍤⍨⍤2|⍳≠ω}⍳t[p]=4]] ⋄ t[i,x]←t[x,i] ⋄ k[i,x]←k[x,i]          ⍝ LG
 n[x]←n[i] ⋄ p←((x,i)@(i,x)⊢⍳≠p)[p]
 n[p⌿⍤⍨(t[p]=2)∧k[p]=3]+←1                                                    ⍝ CI
 p[i]←p[x←p I@{~t[p[ω]]∈3 4}⍤⍨i←⍳t∈4,(⍳3),8+⍳3] ⋄ j←(⌽i)[⍋⌽x]                ⍝ LX
 p t k n r{α[ω]@i⊢α}←⊂j ⋄ p←(i@j⊢⍳≠p)[p]
 s←¯1,⍤⍨∈⍳⍤⍨n[∪x]←⊢∘≠⌸x←0[]⍉e←∪I∘⍋⍤⍨rn←r[b],⍪n[b←⍳t=1]                      ⍝ SL
 d←(≠p)↑d ⋄ d[i←⍳t=3]←0 ⋄ _←{z⍪←d[i]+←ω≠z←r[ω]}⍤⍨⍤≡i ⋄ f←d[0[]⍉e],¯1          ⍝ FR
 xn←n⌿⍤⍨(t=1)∧k[r]=0                                                          ⍝ XN
 v←⍳(t=10)∧n<¯4 ⋄ x←n[y←v,b] ⋄ n[b]←s[e⍳rn] ⋄ i←(≠x)⍴c←≠e                    ⍝ AV
 _←{z/⍤⍨c=i[1[]z]←e⍳⍉x I@1⊢z←r I@0⊢ω}⍤⍨⍤≡(v,r[b])⍪⍉⍪⍳≠x
 f s←(f s I⍤⍤⊂i)⍪@y⍤⊂¯1⍴⍤⍨r ⋄ p t k n f s r d xn sym}
```

APPENDIX C. RACKET NANOPASS SOURCE

```
#lang nanopass

(define (run-compiler ast)
  (foldl (lambda (pass ast) (pass ast)) ast compiler))

(define compiler
  `(,(lambda (x) (record-parent (uniquely-identify x)))
    ,(lambda (x) (record-reference x))
    ,(lambda (x) (lift-functions x))
    ,(lambda (x) (wrap-expressions x))
    ,(lambda (x) (lift-guards x))
    ,(lambda (x) (count-index-children x))
    ,(lambda (x) (lift-expressions x))
    ,(lambda (x) (compute-slots x))
    ,(lambda (x) (compute-frames x))
    ,(lambda (x) (compute-exports x))
    ,(lambda (x) (anchor-variables x))
    ))

(define (name-field? x)
  (eq? x 'n))

(define (reference-field? x)
  (eq? x 'r))

(define (parent-field? x)
  (eq? x 'p))

(define (uid-field? x)
  (eq? x 'uid))

(define (name/ref? x)
  (or (number? x) (symbol? x) (string? x)))

(define-language L_
  (terminals
    (name-field (name))
    (string (ns)))
  (Roots (root)
    (mod* ...))
  (Module (mod)
    (F.0 ([name ns]) stmt* ...))   ;; Module
  (Stmt (stmt) e f)                ;; Function Body Statement
  (Func (f)
    fb
    (P.1 ([name ns]))              ;; Primitive Function
    (V.1 ([name ns]))              ;; Function Variable
    (F.1 ([name ns]) gstmt* ...)   ;; Dfns
```

```
    (O.2 ([name ns]) oprim f)       ;; Monadic Operator
    (O.5 ([name ns]) e oprim f)     ;; AOF Dyadic Operator
    (O.7 ([name ns]) f oprim e)     ;; FOA Dyadic Operator
    (O.8 ([name ns]) f1 oprim f2)) ;; FOF Dyadic Operator
  (GStmt (gstmt) g e fb)
  (Guard (g)
    (G.0 ([name ns]) e1 e2))        ;; Expression Guard
  (FuncBind (fb)
    (B.1 ([name ns]) f))            ;; Function Binding
  (Expr (e)
    v
    (B.0 ([name ns]) e)             ;; Value Binding
    (A.0 ([name ns]) num* ...)      ;; Literal Array
    (A.3 ([name ns]) v* ...)        ;; Stranding; Semi-colon Span
    (E.1 ([name ns]) f e)           ;; Monadic Expression
    (E.2 ([name ns]) e f ebrk)      ;; Dyadic Expr or Bracket Indexing
    (E.4 ([name ns]) v ebrk e))     ;; Assignment
  (BrkOrE (ebrk)
    e
    (E.3 ([name ns]) e* ...))       ;; Bracket Expression
  (Oprim (oprim)
    (P.2 ([name ns])))              ;; Operator Primitive
  (Number (num)
    (N.0 ([name ns])))              ;; Scalar number
  (Value (v)
    (P.0 ([name ns]))               ;; Value Primitive
    (V.0 ([name ns]))))             ;; Value Variable

(define-parser parse L_)

(define-language L0
  (extends L_)
  (terminals
    (- (string (ns)))
    (+ (name/ref (ns)))))

(define-pass symbolify : L_ (ir) -> L0 ()
  (Module : Module (mod) -> Module ()
    [(F.0 ([,name ,ns]) ,[stmt*] ...)
     `(F.0 ([,name ,(string->symbol ns)]) ,stmt* ...)])
  (Func : Func (f) -> Func ()
    [(P.1 ([,name ,ns]))
     `(P.1 ([,name ,(string->symbol ns)]))]
    [(V.1 ([,name ,ns]))
     `(V.1 ([,name ,(string->symbol ns)]))]
    [(F.1 ([,name ,ns]) ,[gstmt*] ...)
     `(F.1 ([,name ,(string->symbol ns)]) ,gstmt* ...)]
    [(O.2 ([,name ,ns]) ,[oprim] ,[f])
     `(O.2 ([,name ,(string->symbol ns)]) ,oprim ,f)]
    [(O.5 ([,name ,ns]) ,[e] ,[oprim] ,[f])
     `(O.5 ([,name ,(string->symbol ns)]) ,e ,oprim ,f)]
```

```
    [(O.7 ([,name ,ns]) ,[f] ,[oprim] ,[e])
     `(O.7 ([,name ,(string->symbol ns)]) ,f ,oprim ,e)]
    [(O.8 ([,name ,ns]) ,[f1] ,[oprim] ,[f2])
     `(O.8 ([,name ,(string->symbol ns)]) ,f1 ,oprim ,f2)])
  (Guard : Guard (g) -> Guard ()
    [(G.0 ([,name ,ns]) ,[e1] ,[e2])
     `(G.0 ([,name ,(string->symbol ns)]) ,e1 ,e2)])
  (FuncBind : FuncBind (fb) -> FuncBind ()
    [(B.1 ([,name ,ns]) ,[f])
     `(B.1 ([,name ,(string->symbol ns)]) ,f)])
  (Expr : Expr (e) -> Expr ()
    [(B.0 ([,name ,ns]) ,[e])
     `(B.0 ([,name ,(string->symbol ns)]) ,e)]
    [(A.0 ([,name ,ns]) ,[num*] ...)
     `(A.0 ([,name ,(string->symbol ns)]) ,num* ...)]
    [(A.3 ([,name ,ns]) ,[v*] ...)
     `(A.3 ([,name ,(string->symbol ns)]) ,v* ...)]
    [(E.1 ([,name ,ns]) ,[f] ,[e])
     `(E.1 ([,name ,(string->symbol ns)]) ,f ,e)]
    [(E.2 ([,name ,ns]) ,[e] ,[f] ,[ebrk])
     `(E.2 ([,name ,(string->symbol ns)]) ,e ,f ,ebrk)]
    [(E.4 ([,name ,ns]) ,[v] ,[ebrk] ,[e])
     `(E.4 ([,name ,(string->symbol ns)]) ,v ,ebrk ,e)])
  (BrkOrE : BrkOrE (ebrk) -> BrkOrE ()
    [(E.3 ([,name ,ns]) ,[e*] ...)
     `(E.3 ([,name ,(string->symbol ns)]) ,e* ...)])
  (Oprim : Oprim (oprim) -> Oprim ()
    [(P.2 ([,name ,ns]))
     `(P.2 ([,name ,(string->symbol ns)]))])
  (Number : Number (num) -> Number ()
    [(N.0 ([,name ,ns]))
     `(N.0 ([,name ,(string->symbol ns)]))])
  (Value : Value (v) -> Value ()
    [(P.0 ([,name ,ns]))
     `(P.0 ([,name ,(string->symbol ns)]))]
    [(V.0 ([,name ,ns]))
     `(V.0 ([,name ,(string->symbol ns)]))]))

(define-language L1
  (terminals
    (name-field (name))
    (uid-field (uid))
    (symbol (id))
    (name/ref (ns)))
  (Roots (root)
    (mod* ...))
  (Module (mod)
    (F.0 ([name ns] [uid id]) stmt* ...))
  (Stmt (stmt) e f)
  (Func (f)
    fb
```

```
    (P.1 ([name ns] [uid id]))
    (V.1 ([name ns] [uid id]))
    (F.1 ([name ns] [uid id]) gstmt* ...)
    (O.2 ([name ns] [uid id]) oprim f)
    (O.5 ([name ns] [uid id]) e oprim f)
    (O.7 ([name ns] [uid id]) f oprim e)
    (O.8 ([name ns] [uid id]) f1 oprim f2))
  (GStmt (gstmt) g e fb)
  (Guard (g)
    (G.0 ([name ns] [uid id]) e1 e2))
  (FuncBind (fb)
    (B.1 ([name ns] [uid id]) f))
  (Expr (e)
    v
    (B.0 ([name ns] [uid id]) e)
    (A.0 ([name ns] [uid id]) num* ...)
    (A.3 ([name ns] [uid id]) v* ...)
    (E.1 ([name ns] [uid id]) f e)
    (E.2 ([name ns] [uid id]) e f ebrk)
    (E.4 ([name ns] [uid id]) v ebrk e))
  (BrkOrE (ebrk)
    e
    (E.3 ([name ns] [uid id]) e* ...))
  (Oprim (oprim)
    (P.2 ([name ns] [uid id])))
  (Number (num)
    (N.0 ([name ns] [uid id])))
  (Value (v)
    (P.0 ([name ns] [uid id]))
    (V.0 ([name ns] [uid id]))))

(define-pass uniquely-identify : L0 (ir) -> L1 ()
  (Module : Module (ir) -> Module ()
    [(F.0 ([,name ,ns]) ,[stmt*] ...)
     `(F.0 ([,name ,ns] [uid ,(gensym "F0:")]) ,stmt* ...)])
  (Func : Func (ir) -> Func ()
    [(P.1 ([,name ,ns]))
     `(P.1 ([,name ,ns] [uid ,(gensym "P1:")]))]
    [(V.1 ([,name ,ns]))
     `(V.1 ([,name ,ns] [uid ,(gensym "V1:")]))]
    [(F.1 ([,name ,ns]) ,[gstmt*] ...)
     `(F.1 ([,name ,ns] [uid ,(gensym "F1:")]) ,gstmt* ...)]
    [(O.2 ([,name ,ns]) ,[oprim] ,[f])
     `(O.2 ([,name ,ns] [uid ,(gensym "O2:")]) ,oprim ,f)]
    [(O.5 ([,name ,ns]) ,[e] ,[oprim] ,[f])
     `(O.5 ([,name ,ns] [uid ,(gensym "O5:")]) ,e ,oprim ,f)]
    [(O.7 ([,name ,ns]) ,[f] ,[oprim] ,[e])
     `(O.7 ([,name ,ns] [uid ,(gensym "O7:")]) ,f ,oprim ,e)]
    [(O.8 ([,name ,ns]) ,[f1] ,[oprim] ,[f2])
     `(O.8 ([,name ,ns] [uid ,(gensym "O8:")]) ,f1 ,oprim ,f2)])
  (Guard : Guard (ir) -> Guard ()
```

```
    [(G.0 ([,name ,ns]) ,[e1] ,[e2])
     `(G.0 ([,name ,ns] [uid ,(gensym "G0:")]) ,e1 ,e2)])
  (FuncBind : FuncBind (ir) -> FuncBind ()
    [(B.1 ([,name ,ns]) ,[f])
     `(B.1 ([,name ,ns] [uid ,(gensym "B1:")]) ,f)])
  (Expr : Expr (ir) -> Expr ()
    [(B.0 ([,name ,ns]) ,[e])
     `(B.0 ([,name ,ns] [uid ,(gensym "B0:")]) ,e)]
    [(A.0 ([,name ,ns]) ,[num*] ...)
     `(A.0 ([,name ,ns] [uid ,(gensym "A0:")]) ,num* ...)]
    [(A.3 ([,name ,ns]) ,[v*] ...)
     `(A.3 ([,name ,ns] [uid ,(gensym "A3:")]) ,v* ...)]
    [(E.1 ([,name ,ns]) ,[f] ,[e])
     `(E.1 ([,name ,ns] [uid ,(gensym "E1:")]) ,f ,e)]
    [(E.2 ([,name ,ns]) ,[e] ,[f] ,[ebrk])
     `(E.2 ([,name ,ns] [uid ,(gensym "E2:")]) ,e ,f ,ebrk)]
    [(E.4 ([,name ,ns]) ,[v] ,[ebrk] ,[e])
     `(E.4 ([,name ,ns] [uidp ,(gensym "E4:")]) ,v ,ebrk ,e)])
  (BrkOrE : BrkOrE (ir) -> BrkOrE ()
    [(E.3 ([,name ,ns]) ,[e*] ...)
     `(E.3 ([,name ,ns] [uid ,(gensym "E3:")]) ,e* ...)])
  (Oprim : Oprim (ir) -> Oprim ()
    [(P.2 ([,name ,ns]))
     `(P.2 ([,name ,ns] [uid ,(gensym "P2:")]))])
  (Number : Number (ir) -> Number ()
    [(N.0 ([,name ,ns]))
     `(N.0 ([,name ,ns] [uid ,(gensym "N0:")]))])
  (Value : Value (ir) -> Value ()
    [(P.0 ([,name ,ns]))
     `(P.0 ([,name ,ns] [uid ,(gensym "P0:")]))]
    [(V.0 ([,name ,ns]))
     `(V.0 ([,name ,ns] [uid ,(gensym "V0:")]))]))

;; Parent Vector
(define-language L2
  (terminals
    (name-field (name))
    (uid-field (uid))
    (parent-field (par))
    (name/ref (ns))
    (symbol (id pid)))
  (Roots (root)
    (mod* ...))
  (Module (mod)
    (F.0 ([name ns] [uid id] [par pid]) stmt* ...))
  (Stmt (stmt) e f)
  (Func (f)
    fb
    (P.1 ([name ns] [uid id] [par pid]))
    (V.1 ([name ns] [uid id] [par pid]))
    (F.1 ([name ns] [uid id] [par pid]) gstmt* ...)
```

```
    (O.2 ([name ns] [uid id] [par pid]) oprim f)
    (O.5 ([name ns] [uid id] [par pid]) e oprim f)
    (O.7 ([name ns] [uid id] [par pid]) f oprim e)
    (O.8 ([name ns] [uid id] [par pid]) f1 oprim f2))
  (GStmt (gstmt) g e fb)
  (Guard (g)
    (G.0 ([name ns] [uid id] [par pid]) e1 e2))
  (FuncBind (fb)
    (B.1 ([name ns] [uid id] [par pid]) f))
  (Expr (e)
    v
    (B.0 ([name ns] [uid id] [par pid]) e)
    (A.0 ([name ns] [uid id] [par pid]) num* ...)
    (A.3 ([name ns] [uid id] [par pid]) v* ...)
    (E.1 ([name ns] [uid id] [par pid]) f e)
    (E.2 ([name ns] [uid id] [par pid]) e f ebrk)
    (E.4 ([name ns] [uid id] [par pid]) v ebrk e))
  (BrkOrE (ebrk)
    e
    (E.3 ([name ns] [uid id] [par pid]) e* ...))
  (Oprim (oprim)
    (P.2 ([name ns] [uid id] [par pid])))
  (Number (num)
    (N.0 ([name ns] [uid id] [par pid])))
  (Value (v)
    (P.0 ([name ns] [uid id] [par pid]))
    (V.0 ([name ns] [uid id] [par pid]))))

(define-pass record-parent : L1 (ir) -> L2 ()
  (Module : Module (ir) -> Module ()
    [(F.0 ([,name ,ns] [,uid ,id]) ,[stmt* id -> stmt*] ...)
     `(F.0 ([,name ,ns] [,uid ,id] [p ,id]) ,stmt* ...)])
  (Stmt : Stmt (ir pid) -> Stmt ())
  (Func : Func (ir pid) -> Func ()
    [(P.1 ([,name ,ns] [,uid ,id]))
     `(P.1 ([,name ,ns] [,uid ,id] [p ,pid]))]
    [(V.1 ([,name ,ns] [,uid ,id]))
     `(V.1 ([,name ,ns] [,uid ,id] [p ,pid]))]
    [(F.1 ([,name ,ns] [,uid ,id]) ,[gstmt* id -> gstmt*] ...)
     `(F.1 ([,name ,ns] [,uid ,id] [p ,pid]) ,gstmt* ...)]
    [(O.2 ([,name ,ns] [,uid ,id]) ,[oprim id -> oprim] ,[f id -> f])
     `(O.2 ([,name ,ns] [,uid ,id] [p ,pid]) ,oprim ,f)]
    [(O.5 ([,name ,ns] [,uid ,id])
       ,[e id -> e] ,[oprim id -> oprim] ,[f id -> f])
     `(O.5 ([,name ,ns] [,uid ,id] [p ,pid]) ,e ,oprim ,f)]
    [(O.7 ([,name ,ns] [,uid ,id])
       ,[f id -> f] ,[oprim id -> oprim] ,[e id -> e])
     `(O.7 ([,name ,ns] [,uid ,id] [p ,pid]) ,f ,oprim ,e)]
    [(O.8 ([,name ,ns] [,uid ,id])
       ,[f1 id -> f1] ,[oprim id -> oprim] ,[f2 id -> f2])
     `(O.8 ([,name ,ns] [,uid ,id] [p ,pid]) ,f1 ,oprim ,f2)])
```

```
  (GStmt : GStmt (ir pid) -> GStmt ())
  (Guard : Guard (ir pid) -> Guard ()
    [(G.0 ([,name ,ns] [,uid ,id]) ,[e1 id -> e1] ,[e2 id -> e2])
     `(G.0 ([,name ,ns] [,uid ,id] [p ,pid]) ,e1 ,e2)])
  (FuncBind : FuncBind (ir pid) -> FuncBind ()
    [(B.1 ([,name ,ns] [,uid ,id]) ,[f id -> f])
     `(B.1 ([,name ,ns] [,uid ,id] [p ,pid]) ,f)])
  (Expr : Expr (ir pid) -> Expr ()
    [(B.0 ([,name ,ns] [,uid ,id]) ,[e id -> e])
     `(B.0 ([,name ,ns] [,uid ,id] [p ,pid]) ,e)]
    [(A.0 ([,name ,ns] [,uid ,id]) ,[num* id -> num*] ...)
     `(A.0 ([,name ,ns] [,uid ,id] [p ,pid]) ,num* ...)]
    [(A.3 ([,name ,ns] [,uid ,id]) ,[v* id -> v*] ...)
     `(A.3 ([,name ,ns] [,uid ,id] [p ,pid]) ,v* ...)]
    [(E.1 ([,name ,ns] [,uid ,id]) ,[f id -> f] ,[e id -> e])
     `(E.1 ([,name ,ns] [,uid ,id] [p ,pid]) ,f ,e)]
    [(E.2 ([,name ,ns] [,uid ,id])
       ,[e id -> e] ,[f id -> f] ,[ebrk id -> ebrk])
     `(E.2 ([,name ,ns] [,uid ,id] [p ,pid]) ,e ,f ,ebrk)]
    [(E.4 ([,name ,ns] [,uid ,id])
       ,[v id -> v] ,[ebrk id -> ebrk] ,[e id -> e])
     `(E.4 ([,name ,ns] [,uid ,id] [p ,pid]) ,v ,ebrk ,e)])
  (BrkOrE : BrkOrE (ir pid) -> BrkOrE ()
    [(E.3 ([,name ,ns] [,uid ,id]) ,[e* id -> e*] ...)
     `(E.3 ([,name ,ns] [,uid ,id] [p ,pid]) ,e* ...)])
  (Oprim : Oprim (ir pid) -> Oprim ()
    [(P.2 ([,name ,ns] [,uid ,id]))
     `(P.2 ([,name ,ns] [,uid ,id] [p ,pid]))])
  (Number : Number (ir pid) -> Number ()
    [(N.0 ([,name ,ns] [,uid ,id]))
     `(N.0 ([,name ,ns] [,uid ,id] [p ,pid]))])
  (Value : Value (ir pid) -> Value ()
    [(P.0 ([,name ,ns] [,uid ,id]))
     `(P.0 ([,name ,ns] [,uid ,id] [p ,pid]))]
    [(V.0 ([,name ,ns] [,uid ,id]))
     `(V.0 ([,name ,ns] [,uid ,id] [p ,pid]))]))

;; Reference Vector
(define-language L3
  (terminals
    (name-field (name))
    (uid-field (uid))
    (parent-field (par))
    (reference-field (ref))
    (name/ref (ns))
    (symbol (id pid rid)))
  (Roots (root)
    (mod* ...))
  (Module (mod)
    (F.0 ([name ns] [uid id] [par pid] [ref rid]) stmt* ...))
  (Stmt (stmt) e f)
```

```
  (Func (f)
    fb
    (P.1 ([name ns] [uid id] [par pid] [ref rid]))
    (V.1 ([name ns] [uid id] [par pid] [ref rid]))
    (F.1 ([name ns] [uid id] [par pid] [ref rid]) gstmt* ...)
    (O.2 ([name ns] [uid id] [par pid] [ref rid]) oprim f)
    (O.5 ([name ns] [uid id] [par pid] [ref rid]) e oprim f)
    (O.7 ([name ns] [uid id] [par pid] [ref rid]) f oprim e)
    (O.8 ([name ns] [uid id] [par pid] [ref rid]) f1 oprim f2))
  (GStmt (gstmt) g e fb)
  (Guard (g)
    (G.0 ([name ns] [uid id] [par pid] [ref rid]) e1 e2))
  (FuncBind (fb)
    (B.1 ([name ns] [uid id] [par pid] [ref rid]) f))
  (Expr (e)
    v
    (B.0 ([name ns] [uid id] [par pid] [ref rid]) e)
    (A.0 ([name ns] [uid id] [par pid] [ref rid]) num* ...)
    (A.3 ([name ns] [uid id] [par pid] [ref rid]) v* ...)
    (E.1 ([name ns] [uid id] [par pid] [ref rid]) f e)
    (E.2 ([name ns] [uid id] [par pid] [ref rid]) e f ebrk)
    (E.4 ([name ns] [uid id] [par pid] [ref rid]) v ebrk e))
  (BrkOrE (ebrk)
    e
    (E.3 ([name ns] [uid id] [par pid] [ref rid]) e* ...))
  (Oprim (oprim)
    (P.2 ([name ns] [uid id] [par pid] [ref rid])))
  (Number (num)
    (N.0 ([name ns] [uid id] [par pid] [ref rid])))
  (Value (v)
    (P.0 ([name ns] [uid id] [par pid] [ref rid]))
    (V.0 ([name ns] [uid id] [par pid] [ref rid]))))

(define-pass record-reference : L2 (ir) -> L3 ()
  (Module : Module (ir) -> Module ()
    [(F.0 ([,name ,ns] [,uid ,id] [,par ,pid])
       ,[stmt* id -> stmt*] ...)
     `(F.0 ([,name ,ns] [,uid ,id] [,par ,pid] [r ,id]) ,stmt* ...)])
  (Stmt : Stmt (ir rid) -> Stmt ())
  (Func : Func (ir rid) -> Func ()
    [(P.1 ([,name ,ns] [,uid ,id] [,par ,pid]))
     `(P.1 ([,name ,ns] [,uid ,id] [,par ,pid] [r ,rid]))]
    [(V.1 ([,name ,ns] [,uid ,id] [,par ,pid]))
     `(V.1 ([,name ,ns] [,uid ,id] [,par ,pid] [r ,rid]))]
    [(F.1 ([,name ,ns] [,uid ,id] [,par ,pid])
       ,[gstmt* id -> gstmt*] ...)
     `(F.1 ([,name ,ns] [,uid ,id] [,par ,pid] [r ,rid]) ,gstmt* ...)]
    [(O.2 ([,name ,ns] [,uid ,id] [,par ,pid]) ,[oprim] ,[f])
     `(O.2 ([,name ,ns] [,uid ,id] [,par ,pid] [r ,rid]) ,oprim ,f)]
    [(O.5 ([,name ,ns] [,uid ,id] [,par ,pid]) ,[e] ,[oprim] ,[f])
     `(O.5 ([,name ,ns] [,uid ,id] [,par ,pid] [r ,rid])
```

```
        ,e ,oprim ,f)]
    [(O.7 ([,name ,ns] [,uid ,id] [,par ,pid]) ,[f] ,[oprim] ,[e])
     `(O.7 ([,name ,ns] [,uid ,id] [,par ,pid] [r ,rid])
        ,f ,oprim ,e)]
    [(O.8 ([,name ,ns] [,uid ,id] [,par ,pid]) ,[f1] ,[oprim] ,[f2])
     `(O.8 ([,name ,ns] [,uid ,id] [,par ,pid] [r ,rid])
        ,f1 ,oprim ,f2)])
  (GStmt : GStmt (ir rid) -> GStmt ())
  (Guard : Guard (ir rid) -> Guard ()
    [(G.0 ([,name ,ns] [,uid ,id] [,par ,pid]) ,[e1] ,[e2])
     `(G.0 ([,name ,ns] [,uid ,id] [,par ,pid] [r ,rid]) ,e1 ,e2)])
  (FuncBind : FuncBind (ir rid) -> FuncBind ()
    [(B.1 ([,name ,ns] [,uid ,id] [,par ,pid]) ,[f])
     `(B.1 ([,name ,ns] [,uid ,id] [,par ,pid] [r ,rid]) ,f)])
  (Expr : Expr (ir rid) -> Expr ()
    [(B.0 ([,name ,ns] [,uid ,id] [,par ,pid]) ,[e])
     `(B.0 ([,name ,ns] [,uid ,id] [,par ,pid] [r ,rid]) ,e)]
    [(A.0 ([,name ,ns] [,uid ,id] [,par ,pid]) ,[num*] ...)
     `(A.0 ([,name ,ns] [,uid ,id] [,par ,pid] [r ,rid]) ,num* ...)]
    [(A.3 ([,name ,ns] [,uid ,id] [,par ,pid]) ,[v*] ...)
     `(A.3 ([,name ,ns] [,uid ,id] [,par ,pid] [r ,rid]) ,v* ...)]
    [(E.1 ([,name ,ns] [,uid ,id] [,par ,pid]) ,[f] ,[e])
     `(E.1 ([,name ,ns] [,uid ,id] [,par ,pid] [r ,rid]) ,f ,e)]
    [(E.2 ([,name ,ns] [,uid ,id] [,par ,pid]) ,[e] ,[f] ,[ebrk])
     `(E.2 ([,name ,ns] [,uid ,id] [,par ,pid] [r ,rid]) ,e ,f ,ebrk)]
    [(E.4 ([,name ,ns] [,uid ,id] [,par ,pid]) ,[v] ,[ebrk] ,[e])
     `(E.4 ([,name ,ns] [,uid ,id] [,par ,pid] [r ,rid])
        ,v ,ebrk ,e)])
  (BrkOrE : BrkOrE (ir rid) -> BrkOrE ()
    [(E.3 ([,name ,ns] [,uid ,id] [,par ,pid]) ,[e*] ...)
     `(E.3 ([,name ,ns] [,uid ,id] [,par ,pid] [r ,rid]) ,e* ...)])
  (Oprim : Oprim (ir rid) -> Oprim ()
    [(P.2 ([,name ,ns] [,uid ,id] [,par ,pid]))
     `(P.2 ([,name ,ns] [,uid ,id] [,par ,pid] [r ,rid]))])
  (Number : Number (ir rid) -> Number ()
    [(N.0 ([,name ,ns] [,uid ,id] [,par ,pid]))
     `(N.0 ([,name ,ns] [,uid ,id] [,par ,pid] [r ,rid]))])
  (Value : Value (ir rid) -> Value ()
    [(P.0 ([,name ,ns] [,uid ,id] [,par ,pid]))
     `(P.0 ([,name ,ns] [,uid ,id] [,par ,pid] [r ,rid]))]
    [(V.0 ([,name ,ns] [,uid ,id] [,par ,pid]))
     `(V.0 ([,name ,ns] [,uid ,id] [,par ,pid] [r ,rid]))]))

;; Lift Functions
(define-language L4
  (extends L3)
  (Module (mod)
    (+ (F.1 ([name ns] [uid id] [par pid] [ref rid]) gstmt* ...)))
  (Func (f)
    (- (F.1 ([name ns] [uid id] [par pid] [ref rid]) gstmt* ...))))
```

```
(define-pass lift-functions : L3 (ir) -> L4 ()
  (Roots : Roots (ir) -> Roots ()
    [(,[mod* mod**] ...)
     (let ([nmod* (append* mod**)])
       `(,mod* ... ,nmod* ...))])
  (Module : Module (ir) -> Module (fn*)
    [(F.0 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
       ,[stmt* fn**] ...)
     (values `(F.0 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
                ,stmt* ...)
             (append* fn**))])
  (Stmt : Stmt (ir) -> Stmt (fn*))
  (Func : Func (ir) -> Func (fn*)
    [(P.1 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]))
     (values `(P.1 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]))
             `())]
    [(V.1 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]))
     (values `(V.1 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]))
             `())]
    [(F.1 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
       ,[gstmt* fn**] ...)
     (let ([f (in-context Module
                `(F.1 ([,name ,ns] [,uid ,id] [,par ,id] [,ref ,rid])
                   ,gstmt* ...))])
       (values `(V.1 ([n ,id]
                      [uid ,(gensym "V1:")]
                      [,par ,pid]
                      [,ref ,rid]))
               (append* (list f) fn**)))]
    [(O.2 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
       ,[oprim] ,[f fn*])
     (values `(O.2 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
                ,oprim ,f)
             fn*)]
    [(O.5 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
       ,[e fn1*] ,[oprim] ,[f fn2*])
     (values `(O.5 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
                ,e ,oprim ,f)
             (append fn1* fn2*))]
    [(O.7 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
       ,[f fn1*] ,[oprim] ,[e fn2*])
     (values `(O.7 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
                ,f ,oprim ,e)
             (append fn1* fn2*))]
    [(O.8 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
       ,[f1 fn1*] ,[oprim] ,[f2 fn2*])
     (values `(O.8 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
                ,f1 ,oprim ,f2)
             (append fn1* fn2*))])
  (GStmt : GStmt (ir) -> GStmt (fn*))
  (Guard : Guard (ir) -> Guard (fn*)
```

```
    [(G.0 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
       ,[e1 f1*] ,[e2 f2*])
     (values `(G.0 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
                ,e1 ,e2)
             (append f1* f2*))])
  (FuncBind : FuncBind (fb) -> FuncBind (fn*)
    [(B.1 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]) ,[f fn*])
     (values `(B.1 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
                ,f)
             fn*)])
  (Expr : Expr (ir) -> Expr (fn*)
    [,v (values (Value v) `())]
    [(B.0 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]) ,[e fn*])
     (values `(B.0 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
                ,e)
             fn*)]
    [(A.0 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
       ,[num*] ...)
     (values `(A.0 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
                ,num* ...)
             `())]
    [(A.3 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]) ,[v*] ...)
     (values `(A.3 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
                ,v* ...)
             `())]
    [(E.1 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
       ,[f fn1*] ,[e fn2*])
     (values `(E.1 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
                ,f ,e)
             (append fn1* fn2*))]
    [(E.2 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
       ,[e fn1*] ,[f fn2*] ,[ebrk fn3*])
     (values `(E.2 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
                ,e ,f ,ebrk)
             (append* (list fn1* fn2* fn3*)))]
    [(E.4 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
       ,[v] ,[ebrk fn1*] ,[e fn2*])
     (values `(E.4 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
                ,v ,ebrk ,e)
             (append fn1* fn2*))])
  (BrkOrE : BrkOrE (ir) -> BrkOrE (fn*)
    [(E.3 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
       ,[e* fn**] ...)
     (values `(E.3 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
                ,e* ...)
             (append* fn**))])
  (Value : Value (ir) -> Value ()))

;; Wrap Expressions
(define-language L5
  (extends L4)
```

```
  (GStmt (gstmt)
    (- e fb)
    (+ wx))
  (Guard (g)
    (- (G.0 ([name ns] [uid id] [par pid] [ref rid]) e1 e2))
    (+ (G.0 ([name ns] [uid id] [par pid] [ref rid]) e wx)))
  (WrapExpr (wx)
    (+ (E.0 ([name ns] [uid id] [par pid] [ref rid]) stmt)
       (E_1 ([name ns] [uid id] [par pid] [ref rid]) stmt)))
  (Module (mod)
    (- (F.0 ([name ns] [uid id] [par pid] [ref rid]) stmt* ...))
    (+ (F.0 ([name ns] [uid id] [par pid] [ref rid]) wx* ...))))

(define-pass wrap-expressions : L4 (ir) -> L5 ()
  (Stmt : Stmt (ir) -> WrapExpr ()
    [,e (WrapExpr e #t)]
    [,f (WrapFunc f)])
  (GStmt : GStmt (ir) -> GStmt ()
    [,e (WrapExpr e #t)]
    [,fb (WrapFunc fb)])
  (Guard : Guard (ir) -> Guard ()
    [(G.0 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
       ,[e1] ,[WrapExpr : e2 #f -> e2])
     `(G.0 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]) ,e1 ,e2)])
  (WrapExpr : Expr (ir func?) -> WrapExpr ()
    (definitions
      (define E0 (gensym "E0:"))
      (define E1 (gensym "E~1:")))
    [,v
     (nanopass-case (L4 Value) v
       [(P.0 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]))
        `(E.0 ([n |0|] [uid ,E0] [p ,pid] [r ,rid])
           (P.0 ([,name ,ns] [,uid ,id] [,par ,E0] [,ref ,rid])))]
       [(V.0 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]))
        `(E.0 ([n |0|] [uid ,E0] [p ,pid] [r ,rid])
           (V.0 ([,name ,ns] [,uid ,id] [,par ,E0] [,ref ,rid])))])]
    [(B.0 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]) ,[e])
     (if func?
         `(E_1 ([n |0|] [uid ,E1] [p ,pid] [r ,rid])
            (B.0 ([,name ,ns] [,uid ,id] [,par ,E1] [,ref ,rid]) ,e))
         `(E.0 ([n |0|] [uid ,E0] [p ,pid] [r ,rid])
            (B.0 ([,name ,ns] [,uid ,id] [,par ,E0] [,ref ,rid])
              ,e)))]
    [(A.0 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
       ,[num*] ...)
     `(E.0 ([n |0|] [uid ,E0] [p ,pid] [r ,rid])
        (A.0 ([,name ,ns] [,uid ,id] [,par ,E0] [,ref ,rid])
          ,num* ...))]
    [(A.3 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]) ,[v*] ...)
     `(E.0 ([n |0|] [uid ,E0] [p ,pid] [r ,rid])
        (A.3 ([,name ,ns] [,uid ,id] [,par ,E0] [,ref ,rid])
```

```
        ,v* ...))]
  [(E.1 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]) ,[f] ,[e])
   `(E.0 ([n |0|] [uid ,E0] [p ,pid] [r ,rid])
      (E.1 ([,name ,ns] [,uid ,id] [,par ,E0] [,ref ,rid])
        ,f ,e))]
  [(E.2 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
     ,[e] ,[f] ,[ebrk])
   `(E.0 ([n |0|] [uid ,E0] [p ,pid] [r ,rid])
      (E.2 ([,name ,ns] [,uid ,id] [,par ,E0] [,ref ,rid])
        ,e ,f ,ebrk))]
  [(E.4 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
     ,[v] ,[ebrk] ,[e])
   `(E.0 ([n |0|] [uid ,E0] [p ,pid] [r ,rid])
      (E.4 ([,name ,ns] [,uid ,id] [,par ,E0] [,ref ,rid])
        ,v ,ebrk ,e))])
(Func : Func (ir) -> Func ())
(WrapFunc : Func (ir) -> WrapExpr ()
  (definitions
    (define sym (gensym "E_1:")))
  [,fb
   (nanopass-case (L4 FuncBind) fb
     [(B.1 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]) ,f)
      `(E_1 ([n |0|] [uid ,sym] [p ,pid] [r ,rid])
         (B.1 ([,name ,ns] [,uid ,id] [,par ,sym] [,ref ,rid])
           ,(Func f)))])]
  [(P.1 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]))
   `(E_1 ([n |0|] [uid ,sym] [p ,pid] [r ,rid])
      (P.1 ([,name ,ns] [,uid ,id] [,par ,sym] [,ref ,rid])))]
  [(V.1 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]))
   `(E_1 ([n |0|] [uid ,sym] [p ,pid] [r ,rid])
      (V.1 ([,name ,ns] [,uid ,id] [,par ,sym] [,ref ,rid])))]
  [(O.2 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
     ,[oprim] ,[f])
   `(E_1 ([n |0|] [uid ,sym] [p ,pid] [r ,rid])
      (O.2 ([,name ,ns] [,uid ,id] [,par ,sym] [,ref ,rid])
        ,oprim ,f))]
  [(O.5 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
     ,[e] ,[oprim] ,[f])
   `(E_1 ([n |0|] [uid ,sym] [p ,pid] [r ,rid])
      (O.5 ([,name ,ns] [,uid ,id] [,par ,sym] [,ref ,rid])
        ,e ,oprim ,f))]
  [(O.7 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
     ,[f] ,[oprim] ,[e])
   `(E_1 ([n |0|] [uid ,sym] [p ,pid] [r ,rid])
      (O.7 ([,name ,ns] [,uid ,id] [,par ,sym] [,ref ,rid])
        ,f ,oprim ,e))]
  [(O.8 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
     ,[f1] ,[oprim] ,[f2])
   `(E_1 ([n |0|] [uid ,sym] [p ,pid] [r ,rid])
      (O.8 ([,name ,ns] [,uid ,id] [,par ,sym] [,ref ,rid])
        ,f1 ,oprim ,f2))]))
```

```
;; Lift Guards
(define-language L6
  (extends L5)
  (GStmt (gstmt)
    (+ e))
  (Guard (g)
    (- (G.0 ([name ns] [uid id] [par pid] [ref rid]) e wx))
    (+ (G.0 ([name ns] [uid id] [par pid] [ref rid]) wx))))

(define-pass lift-guards : L5 (ir) -> L6 ()
  (Module : Module (ir) -> Module ()
    [(F.1 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
       ,[GStmt : gstmt* -> gstmt**] ...)
     (let ([gstmt* (append* gstmt**)])
       `(F.1 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
          ,gstmt* ...))])
  (WrapExpr : WrapExpr (ir) -> WrapExpr ())
  (GStmt : GStmt (ir) -> * ()
    [,wx (list (WrapExpr wx))]
    [,g (let-values ([(g t) (Guard g)]) (list t g))])
  (Guard : Guard (ir) -> Guard (tst)
    [(G.0 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
       ,[LiftExpr : e pid -> e] ,[wx])
     (values `(G.0 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
                ,wx)
             e)])
  (LiftExpr : Expr (ir nid) -> Expr ()
    [(B.0 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]) ,[e])
     `(B.0 ([,name ,ns] [,uid ,id] [,par ,nid] [,ref ,rid]) ,e)]
    [(A.0 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
       ,[num*] ...)
     `(A.0 ([,name ,ns] [,uid ,id] [,par ,nid] [,ref ,rid])
        ,num* ...)]
    [(A.3 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]) ,[v*] ...)
     `(A.3 ([,name ,ns] [,uid ,id] [,par ,nid] [,ref ,rid]) ,v* ...)]
    [(E.1 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]) ,[f] ,[e])
     `(E.1 ([,name ,ns] [,uid ,id] [,par ,nid] [,ref ,rid]) ,f ,e)]
    [(E.2 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
       ,[e] ,[f] ,[ebrk])
     `(E.2 ([,name ,ns] [,uid ,id] [,par ,nid] [,ref ,rid])
        ,e ,f ,ebrk)]
    [(E.4 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
       ,[v] ,[ebrk] ,[e])
     `(E.4 ([,name ,ns] [,uid ,id] [,par ,nid] [,ref ,rid])
        ,v ,ebrk ,e)]))

;; Count Index Children
(define-pass count-index-children : L6 (ir) -> L6 ()
  (Expr : Expr (ir) -> Expr ()
    [(A.3 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]) ,[v*] ...)
```

```
      `(A.3 ([,name ,(length v*)] [,uid ,id] [,par ,pid] [,ref ,rid])
         ,v* ...)])
  (BrkOrE : BrkOrE (ir) -> BrkOrE ()
    [(E.3 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]) ,[e*] ...)
     `(E.3 ([,name ,(length e*)] [,uid ,id] [,par ,pid] [,ref ,rid])
        ,e* ...)]))

;; Lift Expressions
(define-language L7
  (terminals
    (name-field (name))
    (uid-field (uid))
    (parent-field (par))
    (reference-field (ref))
    (name/ref (ns))
    (symbol (id pid rid)))
  (Roots (root)
    (mod* ...))
  (Module (mod)
    (F.1 ([name ns] [uid id] [par pid] [ref rid]) ge* ...)
    (F.0 ([name ns] [uid id] [par pid] [ref rid]) e* ...))
  (GorE (ge) g e)
  (Guard (g)
    (G.0 ([name ns] [uid id] [par pid] [ref rid]) e* ...))
  (Expr (e)
    (P.0 ([name ns] [uid id] [par pid] [ref rid]))
    (V.0 ([name ns] [uid id] [par pid] [ref rid]))
    (P.2 ([name ns] [uid id] [par pid] [ref rid]))
    (E.3 ([name ns] [uid id] [par pid] [ref rid]))
    (B.1 ([name ns] [uid id] [par pid] [ref rid]))
    (P.1 ([name ns] [uid id] [par pid] [ref rid]))
    (V.1 ([name ns] [uid id] [par pid] [ref rid]))
    (O.2 ([name ns] [uid id] [par pid] [ref rid]))
    (O.5 ([name ns] [uid id] [par pid] [ref rid]))
    (O.7 ([name ns] [uid id] [par pid] [ref rid]))
    (O.8 ([name ns] [uid id] [par pid] [ref rid]))
    (E.0 ([name ns] [uid id] [par pid] [ref rid]))
    (E_1 ([name ns] [uid id] [par pid] [ref rid]))
    (B.0 ([name ns] [uid id] [par pid] [ref rid]))
    (A.0 ([name ns] [uid id] [par pid] [ref rid]) num* ...)
    (A.3 ([name ns] [uid id] [par pid] [ref rid]))
    (E.1 ([name ns] [uid id] [par pid] [ref rid]))
    (E.2 ([name ns] [uid id] [par pid] [ref rid]))
    (E.4 ([name ns] [uid id] [par pid] [ref rid])))
  (Number (num)
    (N.0 ([name ns] [uid id] [par pid] [ref rid]))))

(define-pass lift-expressions : L6 (ir) -> L7 ()
  (Module : Module (ir) -> Module ()
    [(F.1 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
       ,gstmt* ...)
```

```
   (let ([ge* (foldr (lambda (gstmt e*) (GorE gstmt e* id))
                     '() gstmt*)])
     `(F.1 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
        ,ge* ...))]
  [(F.0 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]) ,wx* ...)
   (let ([e* (foldr (lambda (wx e*) (WrapExpr wx e* id))
                    '() wx*)])
     `(F.0 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
        ,e* ...))])
(GorE : GStmt (ir e* pid) -> * ()
  [,e (Expr e e* pid)]
  [,wx (WrapExpr wx e* pid)]
  [,g (cons (Guard g) e*)])
(Stmt : Stmt (ir e* pid) -> * ()
  [,e (Expr e e* pid)]
  [,f (Func f e* pid)])
(Guard : Guard (ir) -> Guard ()
  [(G.0 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
     ,[WrapExpr : wx '() id -> e*])
   `(G.0 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]) ,e* ...)])
(WrapExpr : WrapExpr (ir e* nid) -> * ()
  [(E.0 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]) ,stmt)
   (Stmt stmt
         (cons (in-context Expr
                 `(E.0 ([,name ,ns]
                        [,uid ,id]
                        [,par ,nid]
                        [,ref ,rid])))
               e*)
         nid)]
  [(E_1 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]) ,stmt)
   (Stmt stmt
         (cons (in-context Expr
                 `(E_1 ([,name ,ns]
                        [,uid ,id]
                        [,par ,nid]
                        [,ref ,rid])))
               e*)
         nid)])
(Expr : Expr (ir e* nid) -> * ()
  [,v (Value v e* nid)]
  [(B.0 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]) ,e)
   (Expr e
         (cons (in-context Expr
                 `(B.0 ([,name ,ns]
                        [,uid ,id]
                        [,par ,nid]
                        [,ref ,rid])))
               e*)
         nid)]
  [(A.0 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
```

```
      ,[num*] ...)
    (cons (in-context Expr
            `(A.0 ([,name ,ns] [,uid ,id] [,par ,nid] [,ref ,rid])
               ,num* ...))
          e*)]
   [(A.3 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]) ,v* ...)
    (let* ([e* (cons (in-context Expr
                       `(A.3 ([,name ,ns]
                              [,uid ,id]
                              [,par ,nid]
                              [,ref ,rid])))
                     e*)])
      (foldl (lambda (v e*) (Value v e* nid)) e* v*))]
   [(E.1 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]) ,f ,e)
    (let* ([e* (cons (in-context Expr
                       `(E.1 ([,name ,ns]
                              [,uid ,id]
                              [,par ,nid]
                              [,ref ,rid])))
                     e*)]
           [e* (Func f e* nid)])
      (Expr e e* nid))]
   [(E.2 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
      ,e ,f ,ebrk)
    (let* ([e* (cons (in-context Expr
                       `(E.2 ([,name ,ns]
                              [,uid ,id]
                              [,par ,nid]
                              [,ref ,rid])))
                     e*)]
           [e* (Expr e e* nid)]
           [e* (Func f e* nid)])
      (BrkOrE ebrk e* nid))]
   [(E.4 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
      ,v ,ebrk ,e)
    (let* ([e* (cons (in-context Expr
                       `(E.4 ([,name ,ns]
                              [,uid ,id]
                              [,par ,nid]
                              [,ref ,rid])))
                     e*)]
           [e* (Value v e* nid)]
           [e* (BrkOrE ebrk e* nid)])
      (Expr e e* nid))])
 (Func : Func (ir e* nid) -> * ()
   [,fb (FuncBind fb e* nid)]
   [(P.1 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]))
    (cons (in-context Expr
            `(P.1 ([,name ,ns] [,uid ,id] [,par ,nid] [,ref ,rid])))
          e*)]
   [(V.1 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]))
```

```
   (cons (in-context Expr
           `(V.1 ([,name ,ns] [,uid ,id] [,par ,nid] [,ref ,rid])))
         e*)]
  [(O.2 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]) ,oprim ,f)
   (let* ([e* (cons (in-context Expr
                      `(O.2 ([,name ,ns]
                             [,uid ,id]
                             [,par ,nid]
                             [,ref ,rid])))
                    e*)]
          [e* (Oprim oprim e* nid)])
     (Func f e* nid))]
  [(O.5 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
     ,e ,oprim ,f)
   (let* ([e* (cons (in-context Expr
                      `(O.5 ([,name ,ns]
                             [,uid ,id]
                             [,par ,nid]
                             [,ref ,rid])))
                    e*)]
          [e* (Expr e e* nid)]
          [e* (Oprim oprim e* nid)])
     (Func f e* nid))]
  [(O.7 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
     ,f ,oprim ,e)
   (let* ([e* (cons (in-context Expr
                      `(O.7 ([,name ,ns]
                             [,uid ,id]
                             [,par ,nid]
                             [,ref ,rid])))
                    e*)]
          [e* (Func f e* nid)]
          [e* (Oprim oprim e* nid)])
     (Expr e e* nid))]
  [(O.8 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
     ,f1 ,oprim ,f2)
   (let* ([e* (cons (in-context Expr
                      `(O.8 ([,name ,ns]
                             [,uid ,id]
                             [,par ,nid]
                             [,ref ,rid])))
                    e*)]
          [e* (Func f1 e* nid)]
          [e* (Oprim oprim e* nid)])
     (Func f2 e* nid))])
(FuncBind : FuncBind (ir e* nid) -> * ()
  [(B.1 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]) ,f)
   (Func f
         (cons (in-context Expr
                 `(B.1 ([,name ,ns]
                        [,uid ,id]
```

```
                          [,par ,nid]
                          [,ref ,rid])))
                  e*)
          nid)])
  (BrkOrE : BrkOrE (ir te* nid) -> * ()
    [,e (Expr e te* nid)]
    [(E.3 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]) ,e* ...)
     (foldl (lambda (e te*) (Expr e te* nid))
            (cons (in-context Expr
                    `(E.3 ([,name ,ns]
                           [,uid ,id]
                           [,par ,nid]
                           [,ref ,rid])))
                  te*)
            e*)])
  (Oprim : Oprim (ir e* nid) -> * ()
    [(P.2 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]))
     (cons (in-context Expr
             `(P.2 ([,name ,ns] [,uid ,id] [,par ,nid] [,ref ,rid])))
           e*)])
  (Value : Value (ir e* nid) -> * ()
    [(P.0 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]))
     (cons (in-context Expr
             `(P.0 ([,name ,ns] [,uid ,id] [,par ,nid] [,ref ,rid])))
           e*)]
    [(V.0 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]))
     (cons (in-context Expr
             `(V.0 ([,name ,ns] [,uid ,id] [,par ,nid] [,ref ,rid])))
           e*)])
  (Number : Number (ir) -> Number ()))

;; Compute Slots
(define-language L8
  (extends L7)
  (entry Tree+Env)
  (terminals
    (+ (hash (slots frames scopes exports))))
  (Tree+Env (ast)
    (+ (root slots frames scopes exports))))

(define-pass compute-slots : L7 (ir) -> L8 ()
  (Roots : Roots (ir) -> Tree+Env ()
    [(,[mod1 -> mod2] ...)
     (let ([db (make-hasheq (map Module mod1))])
       `((,mod2 ...) ,db ,(hash) ,(hash) ,(hash)))])
  (Module : Module (ir) -> * ()
    [(F.1 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]) ,ge* ...)
     (cons id
           (remove-duplicates (foldr GorE '() ge*)))]
    [(F.0 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]) ,e* ...)
     (cons id
```

```
          (remove-duplicates (foldr Expr '() e*)))])
  (GorE : GorE (ir b*) -> * ()
    [,g (Guard g b*)]
    [,e (Expr e b*)])
  (Guard : Guard (ir b*) -> * ()
    [(G.0 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]) ,e* ...)
     (foldr Expr '() b*)])
  (Expr : Expr (ir b*) -> * ()
    [(B.1 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]))
     (cons ns b*)]
    [(B.0 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]))
     (cons ns b*)]
    [else b*])
  (Roots ir))

;; Compute Frames
(define-pass compute-frames : L8 (ir) -> L8 ()
  (definitions
    (define (calculate-frames db)
      (define (calculate-frame f)
        (let ([x (hash-ref db f)])
          (if (eq? x f)
              0
              (+ 1 (calculate-frame x)))))
      (make-hasheq
        (map (lambda (x) (cons x (calculate-frame x)))
             (hash-keys db)))))
  (Tree+Env : Tree+Env (ir) -> Tree+Env ()
    [(,[Roots : root -> scopes^] ,slots ,frames ,scopes ,exports)
     `(,root ,slots ,(calculate-frames scopes^) ,scopes^ ,exports)])
  (Roots : Roots (ir) -> * ()
    [(,[Module : mod* -> ref*] ...) (make-hasheq ref*)])
  (Module : Module (ir) -> * ()
    [(F.1 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]) ,ge* ...)
     (cons id rid)]
    [(F.0 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]) ,e* ...)
     (cons id rid)]))

;; Compute Exports
(define-pass compute-exports : L8 (ir) -> L8 ()
  (Tree+Env : Tree+Env (ir) -> Tree+Env ()
    [(,[Roots : root -> exports^] ,slots ,frames ,scopes ,exports)
     `(,root ,slots ,frames ,scopes ,exports^)])
  (Roots : Roots (ir) -> * ()
    [(,mod* ...) (make-hasheq (foldr Module '() mod*))])
  (Module : Module (ir x*) -> * ()
    [(F.1 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]) ,ge* ...)
     x*]
    [(F.0 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]) ,e* ...)
     (foldr Expr x* e*)])
  (Expr : Expr (ir x*) -> * ()
```

```
    [(B.1 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]))
     (cons (cons id ns) x*)]
    [(B.0 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]))
     (cons (cons id ns) x*)]
    [else x*]))

;; Anchor Variables to Frame and Slot
(define (frame-field? x)
  (eq? x 'frame))
(define (slot-field? x)
  (eq? x 'slot))

(define-language L9
  (extends L8)
  (terminals
    (+ (frame-field (frame))
       (slot-field (slot))
       (integer (fid sid))))
  (Expr (e)
    (- (V.0 ([name ns] [uid id] [par pid] [ref rid]))
       (B.1 ([name ns] [uid id] [par pid] [ref rid]))
       (V.1 ([name ns] [uid id] [par pid] [ref rid]))
       (B.0 ([name ns] [uid id] [par pid] [ref rid])))
    (+ (V.0 ([name ns] [uid id] [par pid]
             [ref rid] [frame fid] [slot sid]))
       (B.1 ([name ns] [uid id] [par pid]
             [ref rid] [frame fid] [slot sid]))
       (V.1 ([name ns] [uid id] [par pid]
             [ref rid] [frame fid] [slot sid]))
       (B.0 ([name ns] [uid id] [par pid]
             [ref rid] [frame fid] [slot sid])))))

(define-pass anchor-variables : L8 (ir) -> L9 ()
  (definitions
    (define-values (slots frames scopes)
      (nanopass-case (L8 Tree+Env) ir
        [(,root ,slots ,frames ,scopes ,exports)
         (values slots frames scopes)]))
    (define (walk-refs rid)
      (hash-ref scopes rid))
    (define (lookup-frame/slot ns rid)
      (if (memq ns '(α ω ωω αα))
          (values -1 -1)
          (let ([vars (hash-ref slots rid)])
            (let ([sid (index-of vars ns)]
                  [fid (hash-ref frames rid)])
              (if sid
                  (values fid sid)
                  (let ([nid (walk-refs rid)])
                    (if (eq? nid rid)
                        (values fid -1)
```

```
                      (lookup-frame/slot ns nid)))))))))
(Expr : Expr (ir) -> Expr ()
  [(V.0 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]))
   (let-values ([(fid sid) (lookup-frame/slot ns rid)])
     `(V.0 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]
            [frame ,fid] [slot ,sid])))]
  [(B.1 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]))
   (let-values ([(fid ns^) (lookup-frame/slot ns rid)])
     (let-values ([(fid sid)
                   (lookup-frame/slot ns (walk-refs rid))])
       `(B.1 ([,name ,ns^] [,uid ,id] [,par ,pid] [,ref ,rid]
              [frame ,fid] [slot ,sid]))))]
  [(V.1 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]))
   (let-values ([(fid sid) (lookup-frame/slot ns rid)])
     `(V.1 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]
            [frame ,fid] [slot ,sid])))]
  [(B.0 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]))
   (let-values ([(fid ns^) (lookup-frame/slot ns rid)])
     (let-values ([(fid sid)
                   (lookup-frame/slot ns (walk-refs rid))])
       `(B.0 ([,name ,ns^] [,uid ,id] [,par ,pid] [,ref ,rid]
              [frame ,fid] [slot ,sid]))))]))
```

APPENDIX D. CHEZ SCHEME NANOPASS SOURCE

```
#!chezscheme

(library (serial)
  (export
    load-ast
    gen-ast
    time-apply
    time-compiler
    run-benchmark
    get-ast-memory-use
    compiler
    get-frames
    get-slots
    run-compiler)
  (import
    (chezscheme)
    (nanopass))

(define (load-ast file)
  (with-input-from-file file
    (lambda () (read))))

(define (gen-ast size)
  (symbolify (parse (make-list size (load-ast "s1_ast.sexpr")))))

(define (time-apply fn args)
  (define-values (cpu-start real-start)
    (values (cpu-time) (real-time)))
  (define result (fn args))
  (define-values (cpu-end real-end)
    (values (cpu-time) (real-time)))
  (values result (- cpu-end cpu-start) (- real-end real-start) -1))

(define (time-compiler ast)
  (define (time-pass res pass)
    (collect (collect-maximum-generation))
    (let-values ([(z c r g) (time-apply pass (car res))])
      (list z (append (cadr res) `(,r)))))
  (cadr (fold-left time-pass `(,ast ()) compiler)))

(define (run-benchmark max)
  (define (go)
    (map (lambda (x) (time-compiler (gen-ast (expt 2 x))))
         (iota max)))
  (map (lambda iter
         (apply map
                (lambda pass (/ (fold-right + 0 pass) 5.0))
                iter))
```

```
       (go) (go) (go) (go) (go)))

(define (get-ast-memory-use n)
  (collect (collect-maximum-generation))
  (let ([start (current-memory-bytes)])
    (let ([x (gen-ast (expt 2 n))])
      (collect (collect-maximum-generation))
      (let ([end (current-memory-bytes)])
        (cons (exact->inexact (/ (- end start) (expt 2 20)))
              x)))))

(define (run-compiler ast)
  (fold-left (lambda (ast pass) (pass ast)) ast compiler))

(define compiler
  `(,(lambda (x) (record-parent (uniquely-identify x)))
    ,(lambda (x) (record-reference x))
    ,(lambda (x) (lift-functions x))
    ,(lambda (x) (wrap-expressions x))
    ,(lambda (x) (lift-guards x))
    ,(lambda (x) (count-index-children x))
    ,(lambda (x) (lift-expressions x))
    ,(lambda (x) (compute-slots x))
    ,(lambda (x) (compute-frames x))
    ,(lambda (x) (compute-exports x))
    ,(lambda (x) (anchor-variables x))
    ))

(define (name-field? x)
  (eq? x 'n))

(define (reference-field? x)
  (eq? x 'r))

(define (parent-field? x)
  (eq? x 'p))

(define (uid-field? x)
  (eq? x 'uid))

(define (name/ref? x)
  (or (number? x) (symbol? x) (string? x)))

(define-language L_
  (terminals
    (name-field (name))
    (string (ns)))
  (Roots (root)
    (mod* ...))
  (Module (mod)
    (F.0 ([name ns]) stmt* ...))   ;; Module
```

```
  (Stmt (stmt) e f)              ;; Function Body Statement
  (Func (f)
    fb
    (P.1 ([name ns]))              ;; Primitive Function
    (V.1 ([name ns]))              ;; Function Variable
    (F.1 ([name ns]) gstmt* ...)   ;; Dfns
    (O.2 ([name ns]) oprim f)      ;; Monadic Operator
    (O.5 ([name ns]) e oprim f)    ;; AOF Dyadic Operator
    (O.7 ([name ns]) f oprim e)    ;; FOA Dyadic Operator
    (O.8 ([name ns]) f1 oprim f2)) ;; FOF Dyadic Operator
  (GStmt (gstmt) g e fb)
  (Guard (g)
    (G.0 ([name ns]) e1 e2))       ;; Expression Guard
  (FuncBind (fb)
    (B.1 ([name ns]) f))           ;; Function Binding
  (Expr (e)
    v
    (B.0 ([name ns]) e)            ;; Value Binding
    (A.0 ([name ns]) num* ...)     ;; Literal Array
    (A.3 ([name ns]) v* ...)       ;; Stranding; Semi-colon Span
    (E.1 ([name ns]) f e)          ;; Monadic Expression
    (E.2 ([name ns]) e f ebrk)     ;; Dyadic Expr or Bracket Indexing
    (E.4 ([name ns]) v ebrk e))    ;; Assignment
  (BrkOrE (ebrk)
    e
    (E.3 ([name ns]) e* ...))      ;; Bracket Expression
  (Oprim (oprim)
    (P.2 ([name ns])))             ;; Operator Primitive
  (Number (num)
    (N.0 ([name ns])))             ;; Scalar number
  (Value (v)
    (P.0 ([name ns]))              ;; Value Primitive
    (V.0 ([name ns]))))             ;; Value Variable

(define-parser parse L_)

(define-language L0
  (extends L_)
  (terminals
    (- (string (ns)))
    (+ (name/ref (ns)))))

(define-pass symbolify : L_ (ir) -> L0 ()
  (Module : Module (mod) -> Module ()
    [(F.0 ([,name ,ns]) ,[stmt*] ...)
     `(F.0 ([,name ,(string->symbol ns)]) ,stmt* ...)])
  (Func : Func (f) -> Func ()
    [(P.1 ([,name ,ns]))
     `(P.1 ([,name ,(string->symbol ns)]))]
    [(V.1 ([,name ,ns]))
     `(V.1 ([,name ,(string->symbol ns)]))]
```

```
    [(F.1 ([,name ,ns]) ,[gstmt*] ...)
     `(F.1 ([,name ,(string->symbol ns)]) ,gstmt* ...)]
    [(O.2 ([,name ,ns]) ,[oprim] ,[f])
     `(O.2 ([,name ,(string->symbol ns)]) ,oprim ,f)]
    [(O.5 ([,name ,ns]) ,[e] ,[oprim] ,[f])
     `(O.5 ([,name ,(string->symbol ns)]) ,e ,oprim ,f)]
    [(O.7 ([,name ,ns]) ,[f] ,[oprim] ,[e])
     `(O.7 ([,name ,(string->symbol ns)]) ,f ,oprim ,e)]
    [(O.8 ([,name ,ns]) ,[f1] ,[oprim] ,[f2])
     `(O.8 ([,name ,(string->symbol ns)]) ,f1 ,oprim ,f2)])
  (Guard : Guard (g) -> Guard ()
    [(G.0 ([,name ,ns]) ,[e1] ,[e2])
     `(G.0 ([,name ,(string->symbol ns)]) ,e1 ,e2)])
  (FuncBind : FuncBind (fb) -> FuncBind ()
    [(B.1 ([,name ,ns]) ,[f])
     `(B.1 ([,name ,(string->symbol ns)]) ,f)])
  (Expr : Expr (e) -> Expr ()
    [(B.0 ([,name ,ns]) ,[e])
     `(B.0 ([,name ,(string->symbol ns)]) ,e)]
    [(A.0 ([,name ,ns]) ,[num*] ...)
     `(A.0 ([,name ,(string->symbol ns)]) ,num* ...)]
    [(A.3 ([,name ,ns]) ,[v*] ...)
     `(A.3 ([,name ,(string->symbol ns)]) ,v* ...)]
    [(E.1 ([,name ,ns]) ,[f] ,[e])
     `(E.1 ([,name ,(string->symbol ns)]) ,f ,e)]
    [(E.2 ([,name ,ns]) ,[e] ,[f] ,[ebrk])
     `(E.2 ([,name ,(string->symbol ns)]) ,e ,f ,ebrk)]
    [(E.4 ([,name ,ns]) ,[v] ,[ebrk] ,[e])
     `(E.4 ([,name ,(string->symbol ns)]) ,v ,ebrk ,e)])
  (BrkOrE : BrkOrE (ebrk) -> BrkOrE ()
    [(E.3 ([,name ,ns]) ,[e*] ...)
     `(E.3 ([,name ,(string->symbol ns)]) ,e* ...)])
  (Oprim : Oprim (oprim) -> Oprim ()
    [(P.2 ([,name ,ns]))
     `(P.2 ([,name ,(string->symbol ns)]))])
  (Number : Number (num) -> Number ()
    [(N.0 ([,name ,ns]))
     `(N.0 ([,name ,(string->symbol ns)]))])
  (Value : Value (v) -> Value ()
    [(P.0 ([,name ,ns]))
     `(P.0 ([,name ,(string->symbol ns)]))]
    [(V.0 ([,name ,ns]))
     `(V.0 ([,name ,(string->symbol ns)]))]))

(define-language L1
  (terminals
    (name-field (name))
    (uid-field (uid))
    (symbol (id))
    (name/ref (ns)))
  (Roots (root)
```

```
    (mod* ...))
  (Module (mod)
    (F.0 ([name ns] [uid id]) stmt* ...))
  (Stmt (stmt) e f)
  (Func (f)
    fb
    (P.1 ([name ns] [uid id]))
    (V.1 ([name ns] [uid id]))
    (F.1 ([name ns] [uid id]) gstmt* ...)
    (O.2 ([name ns] [uid id]) oprim f)
    (O.5 ([name ns] [uid id]) e oprim f)
    (O.7 ([name ns] [uid id]) f oprim e)
    (O.8 ([name ns] [uid id]) f1 oprim f2))
  (GStmt (gstmt) g e fb)
  (Guard (g)
    (G.0 ([name ns] [uid id]) e1 e2))
  (FuncBind (fb)
    (B.1 ([name ns] [uid id]) f))
  (Expr (e)
    v
    (B.0 ([name ns] [uid id]) e)
    (A.0 ([name ns] [uid id]) num* ...)
    (A.3 ([name ns] [uid id]) v* ...)
    (E.1 ([name ns] [uid id]) f e)
    (E.2 ([name ns] [uid id]) e f ebrk)
    (E.4 ([name ns] [uid id]) v ebrk e))
  (BrkOrE (ebrk)
    e
    (E.3 ([name ns] [uid id]) e* ...))
  (Oprim (oprim)
    (P.2 ([name ns] [uid id])))
  (Number (num)
    (N.0 ([name ns] [uid id])))
  (Value (v)
    (P.0 ([name ns] [uid id]))
    (V.0 ([name ns] [uid id]))))

(define-pass uniquely-identify : L0 (ir) -> L1 ()
  (Module : Module (ir) -> Module ()
    [(F.0 ([,name ,ns]) ,[stmt*] ...)
     `(F.0 ([,name ,ns] [uid ,(gensym "F0:")]) ,stmt* ...)])
  (Func : Func (ir) -> Func ()
    [(P.1 ([,name ,ns]))
     `(P.1 ([,name ,ns] [uid ,(gensym "P1:")]))]
    [(V.1 ([,name ,ns]))
     `(V.1 ([,name ,ns] [uid ,(gensym "V1:")]))]
    [(F.1 ([,name ,ns]) ,[gstmt*] ...)
     `(F.1 ([,name ,ns] [uid ,(gensym "F1:")]) ,gstmt* ...)]
    [(O.2 ([,name ,ns]) ,[oprim] ,[f])
     `(O.2 ([,name ,ns] [uid ,(gensym "O2:")]) ,oprim ,f)]
    [(O.5 ([,name ,ns]) ,[e] ,[oprim] ,[f])
```

```
      `(O.5 ([,name ,ns] [uid ,(gensym "O5:")]) ,e ,oprim ,f)]
    [(O.7 ([,name ,ns]) ,[f] ,[oprim] ,[e])
      `(O.7 ([,name ,ns] [uid ,(gensym "O7:")]) ,f ,oprim ,e)]
    [(O.8 ([,name ,ns]) ,[f1] ,[oprim] ,[f2])
      `(O.8 ([,name ,ns] [uid ,(gensym "O8:")]) ,f1 ,oprim ,f2)])
  (Guard : Guard (ir) -> Guard ()
    [(G.0 ([,name ,ns]) ,[e1] ,[e2])
      `(G.0 ([,name ,ns] [uid ,(gensym "G0:")]) ,e1 ,e2)])
  (FuncBind : FuncBind (ir) -> FuncBind ()
    [(B.1 ([,name ,ns]) ,[f])
      `(B.1 ([,name ,ns] [uid ,(gensym "B1:")]) ,f)])
  (Expr : Expr (ir) -> Expr ()
    [(B.0 ([,name ,ns]) ,[e])
      `(B.0 ([,name ,ns] [uid ,(gensym "B0:")]) ,e)]
    [(A.0 ([,name ,ns]) ,[num*] ...)
      `(A.0 ([,name ,ns] [uid ,(gensym "A0:")]) ,num* ...)]
    [(A.3 ([,name ,ns]) ,[v*] ...)
      `(A.3 ([,name ,ns] [uid ,(gensym "A3:")]) ,v* ...)]
    [(E.1 ([,name ,ns]) ,[f] ,[e])
      `(E.1 ([,name ,ns] [uid ,(gensym "E1:")]) ,f ,e)]
    [(E.2 ([,name ,ns]) ,[e] ,[f] ,[ebrk])
      `(E.2 ([,name ,ns] [uid ,(gensym "E2:")]) ,e ,f ,ebrk)]
    [(E.4 ([,name ,ns]) ,[v] ,[ebrk] ,[e])
      `(E.4 ([,name ,ns] [uidp ,(gensym "E4:")]) ,v ,ebrk ,e)])
  (BrkOrE : BrkOrE (ir) -> BrkOrE ()
    [(E.3 ([,name ,ns]) ,[e*] ...)
      `(E.3 ([,name ,ns] [uid ,(gensym "E3:")]) ,e* ...)])
  (Oprim : Oprim (ir) -> Oprim ()
    [(P.2 ([,name ,ns]))
      `(P.2 ([,name ,ns] [uid ,(gensym "P2:")]))])
  (Number : Number (ir) -> Number ()
    [(N.0 ([,name ,ns]))
      `(N.0 ([,name ,ns] [uid ,(gensym "N0:")]))])
  (Value : Value (ir) -> Value ()
    [(P.0 ([,name ,ns]))
      `(P.0 ([,name ,ns] [uid ,(gensym "P0:")]))]
    [(V.0 ([,name ,ns]))
      `(V.0 ([,name ,ns] [uid ,(gensym "V0:")]))]))

;; Parent Vector
(define-language L2
  (terminals
    (name-field (name))
    (uid-field (uid))
    (parent-field (par))
    (name/ref (ns))
    (symbol (id pid)))
  (Roots (root)
    (mod* ...))
  (Module (mod)
    (F.0 ([name ns] [uid id] [par pid]) stmt* ...))
```

```
  (Stmt (stmt) e f)
  (Func (f)
    fb
    (P.1 ([name ns] [uid id] [par pid]))
    (V.1 ([name ns] [uid id] [par pid]))
    (F.1 ([name ns] [uid id] [par pid]) gstmt* ...)
    (O.2 ([name ns] [uid id] [par pid]) oprim f)
    (O.5 ([name ns] [uid id] [par pid]) e oprim f)
    (O.7 ([name ns] [uid id] [par pid]) f oprim e)
    (O.8 ([name ns] [uid id] [par pid]) f1 oprim f2))
  (GStmt (gstmt) g e fb)
  (Guard (g)
    (G.0 ([name ns] [uid id] [par pid]) e1 e2))
  (FuncBind (fb)
    (B.1 ([name ns] [uid id] [par pid]) f))
  (Expr (e)
    v
    (B.0 ([name ns] [uid id] [par pid]) e)
    (A.0 ([name ns] [uid id] [par pid]) num* ...)
    (A.3 ([name ns] [uid id] [par pid]) v* ...)
    (E.1 ([name ns] [uid id] [par pid]) f e)
    (E.2 ([name ns] [uid id] [par pid]) e f ebrk)
    (E.4 ([name ns] [uid id] [par pid]) v ebrk e))
  (BrkOrE (ebrk)
    e
    (E.3 ([name ns] [uid id] [par pid]) e* ...))
  (Oprim (oprim)
    (P.2 ([name ns] [uid id] [par pid])))
  (Number (num)
    (N.0 ([name ns] [uid id] [par pid])))
  (Value (v)
    (P.0 ([name ns] [uid id] [par pid]))
    (V.0 ([name ns] [uid id] [par pid]))))

(define-pass record-parent : L1 (ir) -> L2 ()
  (Module : Module (ir) -> Module ()
    [(F.0 ([,name ,ns] [,uid ,id]) ,[stmt* id -> stmt*] ...)
     `(F.0 ([,name ,ns] [,uid ,id] [p ,id]) ,stmt* ...)])
  (Stmt : Stmt (ir pid) -> Stmt ())
  (Func : Func (ir pid) -> Func ()
    [(P.1 ([,name ,ns] [,uid ,id]))
     `(P.1 ([,name ,ns] [,uid ,id] [p ,pid]))]
    [(V.1 ([,name ,ns] [,uid ,id]))
     `(V.1 ([,name ,ns] [,uid ,id] [p ,pid]))]
    [(F.1 ([,name ,ns] [,uid ,id]) ,[gstmt* id -> gstmt*] ...)
     `(F.1 ([,name ,ns] [,uid ,id] [p ,pid]) ,gstmt* ...)]
    [(O.2 ([,name ,ns] [,uid ,id]) ,[oprim id -> oprim] ,[f id -> f])
     `(O.2 ([,name ,ns] [,uid ,id] [p ,pid]) ,oprim ,f)]
    [(O.5 ([,name ,ns] [,uid ,id])
       ,[e id -> e] ,[oprim id -> oprim] ,[f id -> f])
     `(O.5 ([,name ,ns] [,uid ,id] [p ,pid]) ,e ,oprim ,f)]
```

```
    [(O.7 ([,name ,ns] [,uid ,id])
       ,[f id -> f] ,[oprim id -> oprim] ,[e id -> e])
     `(O.7 ([,name ,ns] [,uid ,id] [p ,pid]) ,f ,oprim ,e)]
    [(O.8 ([,name ,ns] [,uid ,id])
       ,[f1 id -> f1] ,[oprim id -> oprim] ,[f2 id -> f2])
     `(O.8 ([,name ,ns] [,uid ,id] [p ,pid]) ,f1 ,oprim ,f2)])
  (GStmt : GStmt (ir pid) -> GStmt ())
  (Guard : Guard (ir pid) -> Guard ()
    [(G.0 ([,name ,ns] [,uid ,id]) ,[e1 id -> e1] ,[e2 id -> e2])
     `(G.0 ([,name ,ns] [,uid ,id] [p ,pid]) ,e1 ,e2)])
  (FuncBind : FuncBind (ir pid) -> FuncBind ()
    [(B.1 ([,name ,ns] [,uid ,id]) ,[f id -> f])
     `(B.1 ([,name ,ns] [,uid ,id] [p ,pid]) ,f)])
  (Expr : Expr (ir pid) -> Expr ()
    [(B.0 ([,name ,ns] [,uid ,id]) ,[e id -> e])
     `(B.0 ([,name ,ns] [,uid ,id] [p ,pid]) ,e)]
    [(A.0 ([,name ,ns] [,uid ,id]) ,[num* id -> num*] ...)
     `(A.0 ([,name ,ns] [,uid ,id] [p ,pid]) ,num* ...)]
    [(A.3 ([,name ,ns] [,uid ,id]) ,[v* id -> v*] ...)
     `(A.3 ([,name ,ns] [,uid ,id] [p ,pid]) ,v* ...)]
    [(E.1 ([,name ,ns] [,uid ,id]) ,[f id -> f] ,[e id -> e])
     `(E.1 ([,name ,ns] [,uid ,id] [p ,pid]) ,f ,e)]
    [(E.2 ([,name ,ns] [,uid ,id])
       ,[e id -> e] ,[f id -> f] ,[ebrk id -> ebrk])
     `(E.2 ([,name ,ns] [,uid ,id] [p ,pid]) ,e ,f ,ebrk)]
    [(E.4 ([,name ,ns] [,uid ,id])
       ,[v id -> v] ,[ebrk id -> ebrk] ,[e id -> e])
     `(E.4 ([,name ,ns] [,uid ,id] [p ,pid]) ,v ,ebrk ,e)])
  (BrkOrE : BrkOrE (ir pid) -> BrkOrE ()
    [(E.3 ([,name ,ns] [,uid ,id]) ,[e* id -> e*] ...)
     `(E.3 ([,name ,ns] [,uid ,id] [p ,pid]) ,e* ...)])
  (Oprim : Oprim (ir pid) -> Oprim ()
    [(P.2 ([,name ,ns] [,uid ,id]))
     `(P.2 ([,name ,ns] [,uid ,id] [p ,pid]))])
  (Number : Number (ir pid) -> Number ()
    [(N.0 ([,name ,ns] [,uid ,id]))
     `(N.0 ([,name ,ns] [,uid ,id] [p ,pid]))])
  (Value : Value (ir pid) -> Value ()
    [(P.0 ([,name ,ns] [,uid ,id]))
     `(P.0 ([,name ,ns] [,uid ,id] [p ,pid]))]
    [(V.0 ([,name ,ns] [,uid ,id]))
     `(V.0 ([,name ,ns] [,uid ,id] [p ,pid]))]))

;; Reference Vector
(define-language L3
  (terminals
    (name-field (name))
    (uid-field (uid))
    (parent-field (par))
    (reference-field (ref))
    (name/ref (ns))
```

```
    (symbol (id pid rid)))
  (Roots (root)
    (mod* ...))
  (Module (mod)
    (F.0 ([name ns] [uid id] [par pid] [ref rid]) stmt* ...))
  (Stmt (stmt) e f)
  (Func (f)
    fb
    (P.1 ([name ns] [uid id] [par pid] [ref rid]))
    (V.1 ([name ns] [uid id] [par pid] [ref rid]))
    (F.1 ([name ns] [uid id] [par pid] [ref rid]) gstmt* ...)
    (O.2 ([name ns] [uid id] [par pid] [ref rid]) oprim f)
    (O.5 ([name ns] [uid id] [par pid] [ref rid]) e oprim f)
    (O.7 ([name ns] [uid id] [par pid] [ref rid]) f oprim e)
    (O.8 ([name ns] [uid id] [par pid] [ref rid]) f1 oprim f2))
  (GStmt (gstmt) g e fb)
  (Guard (g)
    (G.0 ([name ns] [uid id] [par pid] [ref rid]) e1 e2))
  (FuncBind (fb)
    (B.1 ([name ns] [uid id] [par pid] [ref rid]) f))
  (Expr (e)
    v
    (B.0 ([name ns] [uid id] [par pid] [ref rid]) e)
    (A.0 ([name ns] [uid id] [par pid] [ref rid]) num* ...)
    (A.3 ([name ns] [uid id] [par pid] [ref rid]) v* ...)
    (E.1 ([name ns] [uid id] [par pid] [ref rid]) f e)
    (E.2 ([name ns] [uid id] [par pid] [ref rid]) e f ebrk)
    (E.4 ([name ns] [uid id] [par pid] [ref rid]) v ebrk e))
  (BrkOrE (ebrk)
    e
    (E.3 ([name ns] [uid id] [par pid] [ref rid]) e* ...))
  (Oprim (oprim)
    (P.2 ([name ns] [uid id] [par pid] [ref rid])))
  (Number (num)
    (N.0 ([name ns] [uid id] [par pid] [ref rid])))
  (Value (v)
    (P.0 ([name ns] [uid id] [par pid] [ref rid]))
    (V.0 ([name ns] [uid id] [par pid] [ref rid]))))

(define-pass record-reference : L2 (ir) -> L3 ()
  (Module : Module (ir) -> Module ()
    [(F.0 ([,name ,ns] [,uid ,id] [,par ,pid])
       ,[stmt* id -> stmt*] ...)
     `(F.0 ([,name ,ns] [,uid ,id] [,par ,pid] [r ,id]) ,stmt* ...)])
  (Stmt : Stmt (ir rid) -> Stmt ())
  (Func : Func (ir rid) -> Func ()
    [(P.1 ([,name ,ns] [,uid ,id] [,par ,pid]))
     `(P.1 ([,name ,ns] [,uid ,id] [,par ,pid] [r ,rid]))]
    [(V.1 ([,name ,ns] [,uid ,id] [,par ,pid]))
     `(V.1 ([,name ,ns] [,uid ,id] [,par ,pid] [r ,rid]))]
    [(F.1 ([,name ,ns] [,uid ,id] [,par ,pid])
```

```
      ,[gstmt* id -> gstmt*] ...)
     `(F.1 ([,name ,ns] [,uid ,id] [,par ,pid] [r ,rid]) ,gstmt* ...)]
    [(O.2 ([,name ,ns] [,uid ,id] [,par ,pid]) ,[oprim] ,[f])
     `(O.2 ([,name ,ns] [,uid ,id] [,par ,pid] [r ,rid]) ,oprim ,f)]
    [(O.5 ([,name ,ns] [,uid ,id] [,par ,pid]) ,[e] ,[oprim] ,[f])
     `(O.5 ([,name ,ns] [,uid ,id] [,par ,pid] [r ,rid])
       ,e ,oprim ,f)]
    [(O.7 ([,name ,ns] [,uid ,id] [,par ,pid]) ,[f] ,[oprim] ,[e])
     `(O.7 ([,name ,ns] [,uid ,id] [,par ,pid] [r ,rid])
       ,f ,oprim ,e)]
    [(O.8 ([,name ,ns] [,uid ,id] [,par ,pid]) ,[f1] ,[oprim] ,[f2])
     `(O.8 ([,name ,ns] [,uid ,id] [,par ,pid] [r ,rid])
       ,f1 ,oprim ,f2)])
  (GStmt : GStmt (ir rid) -> GStmt ())
  (Guard : Guard (ir rid) -> Guard ()
    [(G.0 ([,name ,ns] [,uid ,id] [,par ,pid]) ,[e1] ,[e2])
     `(G.0 ([,name ,ns] [,uid ,id] [,par ,pid] [r ,rid]) ,e1 ,e2)])
  (FuncBind : FuncBind (ir rid) -> FuncBind ()
    [(B.1 ([,name ,ns] [,uid ,id] [,par ,pid]) ,[f])
     `(B.1 ([,name ,ns] [,uid ,id] [,par ,pid] [r ,rid]) ,f)])
  (Expr : Expr (ir rid) -> Expr ()
    [(B.0 ([,name ,ns] [,uid ,id] [,par ,pid]) ,[e])
     `(B.0 ([,name ,ns] [,uid ,id] [,par ,pid] [r ,rid]) ,e)]
    [(A.0 ([,name ,ns] [,uid ,id] [,par ,pid]) ,[num*] ...)
     `(A.0 ([,name ,ns] [,uid ,id] [,par ,pid] [r ,rid]) ,num* ...)]
    [(A.3 ([,name ,ns] [,uid ,id] [,par ,pid]) ,[v*] ...)
     `(A.3 ([,name ,ns] [,uid ,id] [,par ,pid] [r ,rid]) ,v* ...)]
    [(E.1 ([,name ,ns] [,uid ,id] [,par ,pid]) ,[f] ,[e])
     `(E.1 ([,name ,ns] [,uid ,id] [,par ,pid] [r ,rid]) ,f ,e)]
    [(E.2 ([,name ,ns] [,uid ,id] [,par ,pid]) ,[e] ,[f] ,[ebrk])
     `(E.2 ([,name ,ns] [,uid ,id] [,par ,pid] [r ,rid]) ,e ,f ,ebrk)]
    [(E.4 ([,name ,ns] [,uid ,id] [,par ,pid]) ,[v] ,[ebrk] ,[e])
     `(E.4 ([,name ,ns] [,uid ,id] [,par ,pid] [r ,rid])
       ,v ,ebrk ,e)])
  (BrkOrE : BrkOrE (ir rid) -> BrkOrE ()
    [(E.3 ([,name ,ns] [,uid ,id] [,par ,pid]) ,[e*] ...)
     `(E.3 ([,name ,ns] [,uid ,id] [,par ,pid] [r ,rid]) ,e* ...)])
  (Oprim : Oprim (ir rid) -> Oprim ()
    [(P.2 ([,name ,ns] [,uid ,id] [,par ,pid]))
     `(P.2 ([,name ,ns] [,uid ,id] [,par ,pid] [r ,rid]))])
  (Number : Number (ir rid) -> Number ()
    [(N.0 ([,name ,ns] [,uid ,id] [,par ,pid]))
     `(N.0 ([,name ,ns] [,uid ,id] [,par ,pid] [r ,rid]))])
  (Value : Value (ir rid) -> Value ()
    [(P.0 ([,name ,ns] [,uid ,id] [,par ,pid]))
     `(P.0 ([,name ,ns] [,uid ,id] [,par ,pid] [r ,rid]))]
    [(V.0 ([,name ,ns] [,uid ,id] [,par ,pid]))
     `(V.0 ([,name ,ns] [,uid ,id] [,par ,pid] [r ,rid]))]))

;; Lift Functions
(define-language L4
```

```
  (extends L3)
  (Module (mod)
    (+ (F.1 ([name ns] [uid id] [par pid] [ref rid]) gstmt* ...)))
  (Func (f)
    (- (F.1 ([name ns] [uid id] [par pid] [ref rid]) gstmt* ...))))

(define-pass lift-functions : L3 (ir) -> L4 ()
  (Roots : Roots (ir) -> Roots ()
    [(,[mod* mod**] ...)
     (let ([nmod* (apply append mod**)])
       `(,mod* ... ,nmod* ...))])
  (Module : Module (ir) -> Module (fn*)
    [(F.0 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
       ,[stmt* fn**] ...)
     (values `(F.0 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
                ,stmt* ...)
             (apply append fn**))])
  (Stmt : Stmt (ir) -> Stmt (fn*))
  (Func : Func (ir) -> Func (fn*)
    [(P.1 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]))
     (values `(P.1 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]))
             `())]
    [(V.1 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]))
     (values `(V.1 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]))
             `())]
    [(F.1 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
       ,[gstmt* fn**] ...)
     (let ([f (in-context Module
                `(F.1 ([,name ,ns] [,uid ,id] [,par ,id] [,ref ,rid])
                   ,gstmt* ...))])
       (values `(V.1 ([n ,id]
                      [uid ,(gensym "V1:")]
                      [,par ,pid]
                      [,ref ,rid]))
               (apply append (list f) fn**)))]
    [(O.2 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
       ,[oprim] ,[f fn*])
     (values `(O.2 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
                ,oprim ,f)
             fn*)]
    [(O.5 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
       ,[e fn1*] ,[oprim] ,[f fn2*])
     (values `(O.5 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
                ,e ,oprim ,f)
             (append fn1* fn2*))]
    [(O.7 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
       ,[f fn1*] ,[oprim] ,[e fn2*])
     (values `(O.7 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
                ,f ,oprim ,e)
             (append fn1* fn2*))]
    [(O.8 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
```

```
     ,[f1 fn1*] ,[oprim] ,[f2 fn2*])
   (values `(O.8 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
              ,f1 ,oprim ,f2)
           (append fn1* fn2*))])
(GStmt : GStmt (ir) -> GStmt (fn*))
(Guard : Guard (ir) -> Guard (fn*)
  [(G.0 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
     ,[e1 f1*] ,[e2 f2*])
   (values `(G.0 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
              ,e1 ,e2)
           (append f1* f2*))])
(FuncBind : FuncBind (fb) -> FuncBind (fn*)
  [(B.1 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]) ,[f fn*])
   (values `(B.1 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
              ,f)
           fn*)])
(Expr : Expr (ir) -> Expr (fn*)
  [,v (values (Value v) `())]
  [(B.0 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]) ,[e fn*])
   (values `(B.0 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
              ,e)
           fn*)]
  [(A.0 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
     ,[num*] ...)
   (values `(A.0 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
              ,num* ...)
           `())]
  [(A.3 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]) ,[v*] ...)
   (values `(A.3 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
              ,v* ...)
           `())]
  [(E.1 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
     ,[f fn1*] ,[e fn2*])
   (values `(E.1 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
              ,f ,e)
           (append fn1* fn2*))]
  [(E.2 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
     ,[e fn1*] ,[f fn2*] ,[ebrk fn3*])
   (values `(E.2 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
              ,e ,f ,ebrk)
           (append fn1* fn2* fn3*))]
  [(E.4 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
     ,[v] ,[ebrk fn1*] ,[e fn2*])
   (values `(E.4 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
              ,v ,ebrk ,e)
           (append fn1* fn2*))])
(BrkOrE : BrkOrE (ir) -> BrkOrE (fn*)
  [(E.3 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
     ,[e* fn**] ...)
   (values `(E.3 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
              ,e* ...)
```

```
            (apply append fn**))])
  (Value : Value (ir) -> Value ()))

;; Wrap Expressions
(define-language L5
  (extends L4)
  (GStmt (gstmt)
    (- e fb)
    (+ wx))
  (Guard (g)
    (- (G.0 ([name ns] [uid id] [par pid] [ref rid]) e1 e2))
    (+ (G.0 ([name ns] [uid id] [par pid] [ref rid]) e wx)))
  (WrapExpr (wx)
    (+ (E.0 ([name ns] [uid id] [par pid] [ref rid]) stmt)
       (E_1 ([name ns] [uid id] [par pid] [ref rid]) stmt)))
  (Module (mod)
    (- (F.0 ([name ns] [uid id] [par pid] [ref rid]) stmt* ...))
    (+ (F.0 ([name ns] [uid id] [par pid] [ref rid]) wx* ...))))

(define-pass wrap-expressions : L4 (ir) -> L5 ()
  (Stmt : Stmt (ir) -> WrapExpr ()
    [,e (WrapExpr e #t)]
    [,f (WrapFunc f)])
  (GStmt : GStmt (ir) -> GStmt ()
    [,e (WrapExpr e #t)]
    [,fb (WrapFunc fb)])
  (Guard : Guard (ir) -> Guard ()
    [(G.0 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
       ,[e1] ,[WrapExpr : e2 #f -> e2])
     `(G.0 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]) ,e1 ,e2)])
  (WrapExpr : Expr (ir func?) -> WrapExpr ()
    (definitions
      (define E0 (gensym "E0:"))
      (define E1 (gensym "E~1:")))
    [,v
     (nanopass-case (L4 Value) v
       [(P.0 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]))
        `(E.0 ([n |0|] [uid ,E0] [p ,pid] [r ,rid])
           (P.0 ([,name ,ns] [,uid ,id] [,par ,E0] [,ref ,rid])))]
       [(V.0 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]))
        `(E.0 ([n |0|] [uid ,E0] [p ,pid] [r ,rid])
           (V.0 ([,name ,ns] [,uid ,id] [,par ,E0] [,ref ,rid])))])]
    [(B.0 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]) ,[e])
     (if func?
         `(E_1 ([n |0|] [uid ,E1] [p ,pid] [r ,rid])
            (B.0 ([,name ,ns] [,uid ,id] [,par ,E1] [,ref ,rid]) ,e))
         `(E.0 ([n |0|] [uid ,E0] [p ,pid] [r ,rid])
            (B.0 ([,name ,ns] [,uid ,id] [,par ,E0] [,ref ,rid])
              ,e)))]
    [(A.0 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
       ,[num*] ...)
```

```
    `(E.0 ([n |0|] [uid ,E0] [p ,pid] [r ,rid])
       (A.0 ([,name ,ns] [,uid ,id] [,par ,E0] [,ref ,rid])
         ,num* ...))]
   [(A.3 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]) ,[v*] ...)
    `(E.0 ([n |0|] [uid ,E0] [p ,pid] [r ,rid])
       (A.3 ([,name ,ns] [,uid ,id] [,par ,E0] [,ref ,rid])
         ,v* ...))]
   [(E.1 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]) ,[f] ,[e])
    `(E.0 ([n |0|] [uid ,E0] [p ,pid] [r ,rid])
       (E.1 ([,name ,ns] [,uid ,id] [,par ,E0] [,ref ,rid])
         ,f ,e))]
   [(E.2 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
      ,[e] ,[f] ,[ebrk])
    `(E.0 ([n |0|] [uid ,E0] [p ,pid] [r ,rid])
       (E.2 ([,name ,ns] [,uid ,id] [,par ,E0] [,ref ,rid])
         ,e ,f ,ebrk))]
   [(E.4 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
      ,[v] ,[ebrk] ,[e])
    `(E.0 ([n |0|] [uid ,E0] [p ,pid] [r ,rid])
       (E.4 ([,name ,ns] [,uid ,id] [,par ,E0] [,ref ,rid])
         ,v ,ebrk ,e))])
(Func : Func (ir) -> Func ())
(WrapFunc : Func (ir) -> WrapExpr ()
  (definitions
    (define sym (gensym "E_1:")))
  [,fb
   (nanopass-case (L4 FuncBind) fb
     [(B.1 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]) ,f)
      `(E_1 ([n |0|] [uid ,sym] [p ,pid] [r ,rid])
         (B.1 ([,name ,ns] [,uid ,id] [,par ,sym] [,ref ,rid])
           ,(Func f)))])]
   [(P.1 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]))
    `(E_1 ([n |0|] [uid ,sym] [p ,pid] [r ,rid])
       (P.1 ([,name ,ns] [,uid ,id] [,par ,sym] [,ref ,rid])))]
   [(V.1 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]))
    `(E_1 ([n |0|] [uid ,sym] [p ,pid] [r ,rid])
       (V.1 ([,name ,ns] [,uid ,id] [,par ,sym] [,ref ,rid])))]
   [(O.2 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
      ,[oprim] ,[f])
    `(E_1 ([n |0|] [uid ,sym] [p ,pid] [r ,rid])
       (O.2 ([,name ,ns] [,uid ,id] [,par ,sym] [,ref ,rid])
         ,oprim ,f))]
   [(O.5 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
      ,[e] ,[oprim] ,[f])
    `(E_1 ([n |0|] [uid ,sym] [p ,pid] [r ,rid])
       (O.5 ([,name ,ns] [,uid ,id] [,par ,sym] [,ref ,rid])
         ,e ,oprim ,f))]
   [(O.7 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
      ,[f] ,[oprim] ,[e])
    `(E_1 ([n |0|] [uid ,sym] [p ,pid] [r ,rid])
       (O.7 ([,name ,ns] [,uid ,id] [,par ,sym] [,ref ,rid])
```

```
          ,f ,oprim ,e))]
    [(O.8 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
       ,[f1] ,[oprim] ,[f2])
     `(E_1 ([n |0|] [uid ,sym] [p ,pid] [r ,rid])
        (O.8 ([,name ,ns] [,uid ,id] [,par ,sym] [,ref ,rid])
          ,f1 ,oprim ,f2))]))

;; Lift Guards
(define-language L6
  (extends L5)
  (GStmt (gstmt)
    (+ e))
  (Guard (g)
    (- (G.0 ([name ns] [uid id] [par pid] [ref rid]) e wx))
    (+ (G.0 ([name ns] [uid id] [par pid] [ref rid]) wx))))

(define-pass lift-guards : L5 (ir) -> L6 ()
  (Module : Module (ir) -> Module ()
    [(F.1 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
       ,[GStmt : gstmt* -> gstmt**] ...)
     (let ([gstmt* (apply append gstmt**)])
       `(F.1 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
          ,gstmt* ...))])
  (WrapExpr : WrapExpr (ir) -> WrapExpr ())
  (GStmt : GStmt (ir) -> * ()
    [,wx (list (WrapExpr wx))]
    [,g (let-values ([(g t) (Guard g)]) (list t g))])
  (Guard : Guard (ir) -> Guard (tst)
    [(G.0 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
       ,[LiftExpr : e pid -> e] ,[wx])
     (values `(G.0 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
                ,wx)
             e)])
  (LiftExpr : Expr (ir nid) -> Expr ()
    [(B.0 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]) ,[e])
     `(B.0 ([,name ,ns] [,uid ,id] [,par ,nid] [,ref ,rid]) ,e)]
    [(A.0 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
       ,[num*] ...)
     `(A.0 ([,name ,ns] [,uid ,id] [,par ,nid] [,ref ,rid])
        ,num* ...)]
    [(A.3 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]) ,[v*] ...)
     `(A.3 ([,name ,ns] [,uid ,id] [,par ,nid] [,ref ,rid]) ,v* ...)]
    [(E.1 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]) ,[f] ,[e])
     `(E.1 ([,name ,ns] [,uid ,id] [,par ,nid] [,ref ,rid]) ,f ,e)]
    [(E.2 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
       ,[e] ,[f] ,[ebrk])
     `(E.2 ([,name ,ns] [,uid ,id] [,par ,nid] [,ref ,rid])
        ,e ,f ,ebrk)]
    [(E.4 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
       ,[v] ,[ebrk] ,[e])
     `(E.4 ([,name ,ns] [,uid ,id] [,par ,nid] [,ref ,rid])
```

```
        ,v ,ebrk ,e)]))

;; Count Index Children
(define-pass count-index-children : L6 (ir) -> L6 ()
  (Expr : Expr (ir) -> Expr ()
    [(A.3 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]) ,[v*] ...)
     `(A.3 ([,name ,(length v*)] [,uid ,id] [,par ,pid] [,ref ,rid])
        ,v* ...)])
  (BrkOrE : BrkOrE (ir) -> BrkOrE ()
    [(E.3 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]) ,[e*] ...)
     `(E.3 ([,name ,(length e*)] [,uid ,id] [,par ,pid] [,ref ,rid])
        ,e* ...)]))

;; Lift Expressions
(define-language L7
  (terminals
    (name-field (name))
    (uid-field (uid))
    (parent-field (par))
    (reference-field (ref))
    (name/ref (ns))
    (symbol (id pid rid)))
  (Roots (root)
    (mod* ...))
  (Module (mod)
    (F.1 ([name ns] [uid id] [par pid] [ref rid]) ge* ...)
    (F.0 ([name ns] [uid id] [par pid] [ref rid]) e* ...))
  (GorE (ge) g e)
  (Guard (g)
    (G.0 ([name ns] [uid id] [par pid] [ref rid]) e* ...))
  (Expr (e)
    (P.0 ([name ns] [uid id] [par pid] [ref rid]))
    (V.0 ([name ns] [uid id] [par pid] [ref rid]))
    (P.2 ([name ns] [uid id] [par pid] [ref rid]))
    (E.3 ([name ns] [uid id] [par pid] [ref rid]))
    (B.1 ([name ns] [uid id] [par pid] [ref rid]))
    (P.1 ([name ns] [uid id] [par pid] [ref rid]))
    (V.1 ([name ns] [uid id] [par pid] [ref rid]))
    (O.2 ([name ns] [uid id] [par pid] [ref rid]))
    (O.5 ([name ns] [uid id] [par pid] [ref rid]))
    (O.7 ([name ns] [uid id] [par pid] [ref rid]))
    (O.8 ([name ns] [uid id] [par pid] [ref rid]))
    (E.0 ([name ns] [uid id] [par pid] [ref rid]))
    (E_1 ([name ns] [uid id] [par pid] [ref rid]))
    (B.0 ([name ns] [uid id] [par pid] [ref rid]))
    (A.0 ([name ns] [uid id] [par pid] [ref rid]) num* ...)
    (A.3 ([name ns] [uid id] [par pid] [ref rid]))
    (E.1 ([name ns] [uid id] [par pid] [ref rid]))
    (E.2 ([name ns] [uid id] [par pid] [ref rid]))
    (E.4 ([name ns] [uid id] [par pid] [ref rid])))
  (Number (num)
```

```
    (N.0 ([name ns] [uid id] [par pid] [ref rid]))))

(define-pass lift-expressions : L6 (ir) -> L7 ()
  (Module : Module (ir) -> Module ()
    [(F.1 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
       ,gstmt* ...)
     (let ([ge* (fold-right
                  (lambda (gstmt e*) (GorE gstmt e* id))
                  '() gstmt*)])
       `(F.1 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
          ,ge* ...))]
    [(F.0 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]) ,wx* ...)
     (let ([e* (fold-right
                 (lambda (wx e*) (WrapExpr wx e* id))
                 '() wx*)])
       `(F.0 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
          ,e* ...))])
  (GorE : GStmt (ir e* pid) -> * ()
    [,e (Expr e e* pid)]
    [,wx (WrapExpr wx e* pid)]
    [,g (cons (Guard g) e*)])
  (Stmt : Stmt (ir e* pid) -> * ()
    [,e (Expr e e* pid)]
    [,f (Func f e* pid)])
  (Guard : Guard (ir) -> Guard ()
    [(G.0 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
       ,[WrapExpr : wx '() id -> e*])
     `(G.0 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]) ,e* ...)])
  (WrapExpr : WrapExpr (ir e* nid) -> * ()
    [(E.0 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]) ,stmt)
     (Stmt stmt
           (cons (in-context Expr
                   `(E.0 ([,name ,ns]
                          [,uid ,id]
                          [,par ,nid]
                          [,ref ,rid])))
                 e*)
           nid)]
    [(E_1 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]) ,stmt)
     (Stmt stmt
           (cons (in-context Expr
                   `(E_1 ([,name ,ns]
                          [,uid ,id]
                          [,par ,nid]
                          [,ref ,rid])))
                 e*)
           nid)])
  (Expr : Expr (ir e* nid) -> * ()
    [,v (Value v e* nid)]
    [(B.0 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]) ,e)
     (Expr e
```

```
       (cons (in-context Expr
               `(B.0 ([,name ,ns]
                      [,uid ,id]
                      [,par ,nid]
                      [,ref ,rid])))
             e*)
       nid)]
[(A.0 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
   ,[num*] ...)
 (cons (in-context Expr
         `(A.0 ([,name ,ns] [,uid ,id] [,par ,nid] [,ref ,rid])
           ,num* ...))
       e*)]
[(A.3 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]) ,v* ...)
 (let* ([e* (cons (in-context Expr
                    `(A.3 ([,name ,ns]
                           [,uid ,id]
                           [,par ,nid]
                           [,ref ,rid])))
                  e*)])
   (fold-left (lambda (e* v) (Value v e* nid)) e* v*))]
[(E.1 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]) ,f ,e)
 (let* ([e* (cons (in-context Expr
                    `(E.1 ([,name ,ns]
                           [,uid ,id]
                           [,par ,nid]
                           [,ref ,rid])))
                  e*)]
        [e* (Func f e* nid)])
   (Expr e e* nid))]
[(E.2 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
   ,e ,f ,ebrk)
 (let* ([e* (cons (in-context Expr
                    `(E.2 ([,name ,ns]
                           [,uid ,id]
                           [,par ,nid]
                           [,ref ,rid])))
                  e*)]
        [e* (Expr e e* nid)]
        [e* (Func f e* nid)])
   (BrkOrE ebrk e* nid))]
[(E.4 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
   ,v ,ebrk ,e)
 (let* ([e* (cons (in-context Expr
                    `(E.4 ([,name ,ns]
                           [,uid ,id]
                           [,par ,nid]
                           [,ref ,rid])))
                  e*)]
        [e* (Value v e* nid)]
        [e* (BrkOrE ebrk e* nid)])
```

```
     (Expr e e* nid))])
(Func : Func (ir e* nid) -> * ()
  [,fb (FuncBind fb e* nid)]
  [(P.1 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]))
   (cons (in-context Expr
           `(P.1 ([,name ,ns] [,uid ,id] [,par ,nid] [,ref ,rid])))
         e*)]
  [(V.1 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]))
   (cons (in-context Expr
           `(V.1 ([,name ,ns] [,uid ,id] [,par ,nid] [,ref ,rid])))
         e*)]
  [(O.2 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]) ,oprim ,f)
   (let* ([e* (cons (in-context Expr
                      `(O.2 ([,name ,ns]
                             [,uid ,id]
                             [,par ,nid]
                             [,ref ,rid])))
                    e*)]
          [e* (Oprim oprim e* nid)])
     (Func f e* nid))]
  [(O.5 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
     ,e ,oprim ,f)
   (let* ([e* (cons (in-context Expr
                      `(O.5 ([,name ,ns]
                             [,uid ,id]
                             [,par ,nid]
                             [,ref ,rid])))
                    e*)]
          [e* (Expr e e* nid)]
          [e* (Oprim oprim e* nid)])
     (Func f e* nid))]
  [(O.7 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
     ,f ,oprim ,e)
   (let* ([e* (cons (in-context Expr
                      `(O.7 ([,name ,ns]
                             [,uid ,id]
                             [,par ,nid]
                             [,ref ,rid])))
                    e*)]
          [e* (Func f e* nid)]
          [e* (Oprim oprim e* nid)])
     (Expr e e* nid))]
  [(O.8 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid])
     ,f1 ,oprim ,f2)
   (let* ([e* (cons (in-context Expr
                      `(O.8 ([,name ,ns]
                             [,uid ,id]
                             [,par ,nid]
                             [,ref ,rid])))
                    e*)]
          [e* (Func f1 e* nid)]
```

```
           [e* (Oprim oprim e* nid)])
      (Func f2 e* nid))])
  (FuncBind : FuncBind (ir e* nid) -> * ()
    [(B.1 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]) ,f)
     (Func f
           (cons (in-context Expr
                   `(B.1 ([,name ,ns]
                          [,uid ,id]
                          [,par ,nid]
                          [,ref ,rid])))
                 e*)
           nid)])
  (BrkOrE : BrkOrE (ir te* nid) -> * ()
    [,e (Expr e te* nid)]
    [(E.3 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]) ,e* ...)
     (fold-left
       (lambda (te* e) (Expr e te* nid))
       (cons (in-context Expr
               `(E.3 ([,name ,ns]
                      [,uid ,id]
                      [,par ,nid]
                      [,ref ,rid])))
             te*)
       e*)])
  (Oprim : Oprim (ir e* nid) -> * ()
    [(P.2 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]))
     (cons (in-context Expr
             `(P.2 ([,name ,ns] [,uid ,id] [,par ,nid] [,ref ,rid])))
           e*)])
  (Value : Value (ir e* nid) -> * ()
    [(P.0 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]))
     (cons (in-context Expr
             `(P.0 ([,name ,ns] [,uid ,id] [,par ,nid] [,ref ,rid])))
           e*)]
    [(V.0 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]))
     (cons (in-context Expr
             `(V.0 ([,name ,ns] [,uid ,id] [,par ,nid] [,ref ,rid])))
           e*)])
  (Number : Number (ir) -> Number ()))

;; Compute Slots
(define-language L8
  (extends L7)
  (entry Tree+Env)
  (terminals
    (+ (hashtable (slots frames scopes exports))))
  (Tree+Env (ast)
    (+ (root slots frames scopes exports))))

(define (remove-duplicates lst)
  (let ([lst (sort (lambda (x y)
```

```
                    (string<? (symbol->string x) (symbol->string y)))
                  lst)])
    (fold-right
      (lambda (x y)
        (cond
          [(null? y) (cons x y)]
          [(eq? x (car y)) y]
          [else (cons x y)]))
      '()
      lst)))

(define-pass compute-slots : L7 (ir) -> L8 ()
  (Roots : Roots (ir) -> Tree+Env ()
    [(,[mod1 -> mod2] ...)
     (let ([db (make-eq-hashtable)])
       (for-each (lambda (x) (hashtable-set! db (car x) (cdr x)))
                 (map Module mod1))
       `((,mod2 ...)
         ,db
         ,(make-eq-hashtable)
         ,(make-eq-hashtable)
         ,(make-eq-hashtable)))])
  (Module : Module (ir) -> * ()
    [(F.1 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]) ,ge* ...)
     (cons id
           (remove-duplicates (fold-right GorE '() ge*)))]
    [(F.0 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]) ,e* ...)
     (cons id
           (remove-duplicates (fold-right Expr '() e*)))])
  (GorE : GorE (ir b*) -> * ()
    [,g (Guard g b*)]
    [,e (Expr e b*)])
  (Guard : Guard (ir b*) -> * ()
    [(G.0 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]) ,e* ...)
     (fold-right Expr '() b*)])
  (Expr : Expr (ir b*) -> * ()
    [(B.1 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]))
     (cons ns b*)]
    [(B.0 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]))
     (cons ns b*)]
    [else b*])
  (Roots ir))

;; Compute Frames
(define-pass compute-frames : L8 (ir) -> L8 ()
  (definitions
    (define (calculate-frames db)
      (define (calculate-frame f)
        (let ([x (hashtable-ref db f #f)])
          (cond
            [(not x) (error 'compute-frames "Bad frame lookup.")]
```

```
            [(eq? x f) 0]
            [else (+ 1 (calculate-frame x))])))
      (let ([db2 (make-eq-hashtable)])
        (vector-for-each
          (lambda (x) (hashtable-set! db2 x (calculate-frame x)))
          (hashtable-keys db))
        db2)))
  (Tree+Env : Tree+Env (ir) -> Tree+Env ()
    [(,[Roots : root -> scopes^] ,slots ,frames ,scopes ,exports)
     `(,root ,slots ,(calculate-frames scopes^) ,scopes^ ,exports)])
  (Roots : Roots (ir) -> * ()
    [(,[Module : mod* -> ref*] ...)
     (let ([db (make-eq-hashtable)])
       (for-each (lambda (x) (hashtable-set! db (car x) (cdr x)))
                 ref*)
       db)])
  (Module : Module (ir) -> * ()
    [(F.1 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]) ,ge* ...)
     (cons id rid)]
    [(F.0 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]) ,e* ...)
     (cons id rid)]))

;; Compute Exports
(define-pass compute-exports : L8 (ir) -> L8 ()
  (Tree+Env : Tree+Env (ir) -> Tree+Env ()
    [(,[Roots : root -> exports^] ,slots ,frames ,scopes ,exports)
     `(,root ,slots ,frames ,scopes ,exports^)])
  (Roots : Roots (ir) -> * ()
    [(,mod* ...)
     (let ([db (make-eq-hashtable)])
       (for-each (lambda (x) (hashtable-set! db (car x) (cdr x)))
         (fold-right Module '() mod*))
       db)])
  (Module : Module (ir x*) -> * ()
    [(F.1 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]) ,ge* ...)
     x*]
    [(F.0 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]) ,e* ...)
     (fold-right Expr x* e*)])
  (Expr : Expr (ir x*) -> * ()
    [(B.1 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]))
     (cons (cons id ns) x*)]
    [(B.0 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]))
     (cons (cons id ns) x*)]
    [else x*]))

;; Anchor Variables to Frame and Slot
(define (frame-field? x)
  (eq? x 'frame))
(define (slot-field? x)
  (eq? x 'slot))
```

```
(define-language L9
  (extends L8)
  (terminals
    (+ (frame-field (frame))
       (slot-field (slot))
       (integer (fid sid))))
  (Expr (e)
    (- (V.0 ([name ns] [uid id] [par pid] [ref rid]))
       (B.1 ([name ns] [uid id] [par pid] [ref rid]))
       (V.1 ([name ns] [uid id] [par pid] [ref rid]))
       (B.0 ([name ns] [uid id] [par pid] [ref rid])))
    (+ (V.0 ([name ns] [uid id] [par pid]
             [ref rid] [frame fid] [slot sid]))
       (B.1 ([name ns] [uid id] [par pid]
             [ref rid] [frame fid] [slot sid]))
       (V.1 ([name ns] [uid id] [par pid]
             [ref rid] [frame fid] [slot sid]))
       (B.0 ([name ns] [uid id] [par pid]
             [ref rid] [frame fid] [slot sid])))))

(define (get-frames ast)
  (nanopass-case (L8 Tree+Env) ast
    [(,root ,slots ,frames ,scopes ,exports) frames]))

(define (get-slots ast)
   (nanopass-case (L8 Tree+Env) ast
     [(,root ,slots ,frames ,scopes ,exports) slots]))

(define-pass anchor-variables : L8 (ir) -> L9 ()
  (definitions
    (define (index-of lst key)
      (let loop ([lst lst] [i 0])
        (cond
          [(null? lst) #f]
          [(eqv? key (car lst)) i]
          [else (loop (cdr lst) (+ 1 i))])))
    (define-values (slots frames scopes)
      (nanopass-case (L8 Tree+Env) ir
        [(,root ,slots ,frames ,scopes ,exports)
         (values slots frames scopes)]))
    (define (walk-refs rid)
      (hashtable-ref scopes rid #f))
    (define (lookup-frame/slot ns rid)
      (if (memq ns '(α ω ωω αα))
          (values -1 -1)
          (let ([vars (hashtable-ref slots rid #f)])
            (assert vars)
            (let ([sid (index-of vars ns)]
                  [fid (hashtable-ref frames rid #f)])
              (assert fid)
              (if sid
```

```
                  (values fid sid)
                  (let ([nid (walk-refs rid)])
                    (assert nid)
                    (if (eq? nid rid)
                        (values fid -1)
                        (lookup-frame/slot ns nid)))))))))
  (Expr : Expr (ir) -> Expr ()
    [(V.0 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]))
     (let-values ([(fid sid) (lookup-frame/slot ns rid)])
       `(V.0 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]
              [frame ,fid] [slot ,sid])))]
    [(B.1 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]))
     (let-values ([(fid ns^) (lookup-frame/slot ns rid)])
       (let-values ([(fid sid)
                     (lookup-frame/slot ns (walk-refs rid))])
         `(B.1 ([,name ,ns^] [,uid ,id] [,par ,pid] [,ref ,rid]
                [frame ,fid] [slot ,sid]))))]
    [(V.1 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]))
     (let-values ([(fid sid) (lookup-frame/slot ns rid)])
       `(V.1 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]
              [frame ,fid] [slot ,sid])))]
    [(B.0 ([,name ,ns] [,uid ,id] [,par ,pid] [,ref ,rid]))
     (let-values ([(fid ns^) (lookup-frame/slot ns rid)])
       (let-values ([(fid sid)
                     (lookup-frame/slot ns (walk-refs rid))])
         `(B.0 ([,name ,ns^] [,uid ,id] [,par ,pid] [,ref ,rid]
                [frame ,fid] [slot ,sid]))))]))

)
```

APPENDIX E. BENCHMARK INPUT SOURCE

```
:Namespace s1

r←0.02    ⋄ v←0.03

Run←{S←0⌷⍵ ⋄ X←1⌷⍵ ⋄ T←⍺ ⋄ vsqrtT←v×T*0.5
 L←|(((⊛S÷X)+(r+(v*2)÷2)×T)÷vsqrtT)-vsqrtT
 (÷(○2)*0.5)×(*(L×L)÷¯2)×÷1+0.2316419×L}

coeff←0.31938153 ¯0.356563782 1.781477937 ¯1.821255978 1.33027442
CNDP2←{L←|⍵ ⋄ B←⍵≥0
 R←(÷(○2)*0.5)×(*(L×L)÷¯2)×{coeff+.×⍵*1+⍳5}¨÷1+0.2316419×L
 (1 ¯1)[B]×((0 ¯1)[B])+R
}
bs←{S←0⌷⍵ ⋄ X←1⌷⍵ ⋄ T←⍺ ⋄ vsqrtT←v×T*0.5
 D1←((⊛S÷X)+(r+(v*2)÷2)×T)÷vsqrtT ⋄ D2←D1-vsqrtT
 CD1←CNDP2 D1 ⋄ CD2←CNDP2 D2 ⋄ e←*(-r)×T
 ((S×CD1)-X×e×CD2),[0.5](X×e×1-CD2)-S×1-CD1
}

bs1←{S←0⌷⍵ ⋄ X←1⌷⍵ ⋄ T←⍺ ⋄ vsqrtT←v×T*0.5
 ((⊛S÷X)+(r+(v*2)÷2)×T)÷vsqrtT}

bs2←{S←0⌷⍵ ⋄ X←1⌷⍵ ⋄ T←⍺ ⋄ vsqrtT←v×T*0.5
 D1←((⊛S÷X)+(r+(v*2)÷2)×T)÷vsqrtT ⋄ D2←D1-vsqrtT ⋄ CNDP2 D1}

CNDP2Δ1←{L←|⍵ ⋄ B←⍵≥0
 (÷(○2)*0.5)×(*(L×L)÷¯2)×{coeff+.×⍵*1+⍳5}¨÷1+0.2316419×L}
bs3←{S←0⌷⍵ ⋄ X←1⌷⍵ ⋄ T←⍺ ⋄ vsqrtT←v×T*0.5
 D1←((⊛S÷X)+(r+(v*2)÷2)×T)÷vsqrtT ⋄ D2←D1-vsqrtT ⋄ CNDP2Δ1 D1}

CNDP2Δ2←{L←|⍵ ⋄ B←⍵≥0 ⋄ ÷1+0.2316419×L}
bs4←{S←0⌷⍵ ⋄ X←1⌷⍵ ⋄ T←⍺ ⋄ vsqrtT←v×T*0.5
 D1←((⊛S÷X)+(r+(v*2)÷2)×T)÷vsqrtT ⋄ D2←D1-vsqrtT ⋄ CNDP2Δ2 D1}

CNDP2Δ3←{L←|⍵ ⋄ B←⍵≥0 ⋄ {1+⍵}¨÷1+0.2316419×L}
bs5←{S←0⌷⍵ ⋄ X←1⌷⍵ ⋄ T←⍺ ⋄ vsqrtT←v×T*0.5
 D1←((⊛S÷X)+(r+(v*2)÷2)×T)÷vsqrtT ⋄ D2←D1-vsqrtT ⋄ CNDP2Δ3 D1}

bs6←{coeff+.×⍵}

bs7←{{coeff+.×⍵*1 2 3 4 5}¨⍵}

CNDP2Δ4←{L←|⍵ ⋄ B←⍵≥0 ⋄ {coeff+.×⍵*1+⍳5}¨÷1+0.2316419×L}
bs8←{S←0⌷⍵ ⋄ X←1⌷⍵ ⋄ T←⍺ ⋄ vsqrtT←v×T*0.5
 D1←((⊛S÷X)+(r+(v*2)÷2)×T)÷vsqrtT ⋄ D2←D1-vsqrtT ⋄ CNDP2Δ4 D1}

CNDP2Δ5←{L←|⍵ ⋄ B←⍵≥0 ⋄ 5×{coeff+.×⍵*1+⍳5}¨÷1+0.2316419×L}
bs9←{S←0⌷⍵ ⋄ X←1⌷⍵ ⋄ T←⍺ ⋄ vsqrtT←v×T*0.5
```

```
 D1←((⍟S÷X)+(r+(v*2)÷2)×T)÷vsqrtT ⋄ D2←D1-vsqrtT ⋄ CNDP2 D1}

X←1

R1←{⍵: 0 ⋄ 1} ⋄ R2←{~⍵: 0 ⋄ 1} ⋄ R3←{~⍵∨⍵: 0 ⋄ 1}
R4←{X: 0 ⋄ 1} ⋄ R5←{X←~⍵ ⋄ X:0 ⋄ 1}

:EndNamespace
```

APPENDIX F. RAW SPEEDUPS

```
⍝ Co-dfns GPU vs. CPU
0.039 0.041 0.087 0      0.099 0     0     0      0.165  0     0     0
0.04  0.05  0.072 0      0.09  0     0     0      0      0     0     0.031
0.062 0.029 0.075 0      0.162 0     0     0.231  0.147  0     0     0.059
0.087 0.065 0     0      0.453 0     0     0.079  0.299  0     0     0.06
0.152 0.095 0.117 0      0.74  0.277 0     0.293  0.283  0     0     0.1
0.265 0.176 0.308 0      1.313 0     0.241 0.814  0.08   0     1.06  0.05
0.524 0.38  0.472 0.748  2.31  0     0.435 1.408  0.486  0     0     0.158
0.87  0.629 0.779 0.721  3.798 0.452 0.392 1.997  0.763  0.412 0     0.248
1.402 1.038 1.015 1.399  6.242 0.41  0.73  2.408  0.799  0.204 1.114 0.428
2.136 1.568 1.415 2.391  8.205 1.118 0.881 3.108  1.261  0.629 1.294 0.776
2.885 1.957 1.948 3.873 10.31  1.876 1.279 3.946  1.983  0.855 1.313 1.186
3.845 2.779 2.524 4.154 11.879 2.552 1.811 4.58   3.276  1.943 2.173 1.737
4.574 3.634 2.828 5.489 13.12  2.977 2.369 4.879  5.314  2.921 2.671 2.301
5.477 4.321 3.103 5.492 14.465 3.46  2.788 5.339  9.444 33.864 3.149 3.544
5.963 4.764 3.318 8.786 17.282 3.737 3.017 5.475 13.568  5.007 3.152 4.045

⍝ Co-dfns CPU vs. Racket Nanopass
10.6    4       4      120      7      140      40     120       7       1       1     120
10.167  3.333   5      100      5      200      100    100     100       1       1      11
11.8   10       8      240      6      240      240      5.5    13      40       1       8
15.267  7.8   380      440      5.2    480      420     21      12      20       1      16
15.931  9.75   19.5    820      5.889   47      880     14.333  25      20      20      16.25
17.926 12.875  15.2   1620      6.313 1860       86      7.909  99      20       1.5    67.5
20.328 11.263  21.889   54.25   8.097 4680      110     11.35   51     100      60      45.857
23.347 12.229  24.722  119.5    9.049  256      243.5   12.6    71.375   3.5   120      57.917
27     15.162  30.133  106.667  9.714  534      254.75  13.507 109      16       3.667  69.714
29.232 23.683  29.016   98.35   9.954  308.143 339.5    13.019 113.714   7.5     4.6    64.13
27.744 23.178  26.035   81.917  9.786  240.667 319.462  11.344 125.868   9.143   5.875  59.44
27.013 23.519  22.914   92.965  9.787  218.175 287.103  10.394 139.058  10.467   5      57.721
34.705 21.273  21.488   78.365 14.73   344.531 421.203  15.674 198.141  16.034   5.974  75.563
32.924 22.914  21.191   90.626 14.72   327.116 419.603  17.187 156.27    2.252   8.188  53.441
33.433 21.442  21.258   60.79  14.322  369.242 458.082  17.677 136.372  19.892   8.658  50.921

⍝ Co-dfns GPU vs. Racket Nanopass
  0.417   0.164  0.347   1.587   0.695    2.499    0.575  0.775    1.157  0.012  0.033   0.222
  0.404   0.165  0.361   1.215   0.449    3.049    1.225  0.583    0.764  0.011  0.028   0.339
  0.726   0.29   0.596   2.526   0.972    3.432    2.841  1.273    1.907  0.455  0.028   0.474
  1.332   0.511  1.347   5.05    2.358    7.679    5.281  1.666    3.587  0.228  0.026   0.959
  2.423   0.921  2.286   8.788   4.356   13.022    9.738  4.205    7.072  0.204  0.524   1.629
  4.743   2.264  4.686  17.341   8.288   24.074   20.711  6.441    7.871  0.227  1.59    3.403
 10.646   4.284 10.342  40.576  18.707   60.581   47.856 15.979   24.785  1.076  1.526   7.259
 20.323   7.695 19.251  86.111  34.371  115.761   95.385 25.157   54.467  1.44   2.661  14.345
 37.843  15.746 30.585 149.261  60.638  218.987  186.058 32.522   87.094  3.257  4.086  29.823
 62.443  37.139 41.06  235.144  81.672  344.525  299.123 40.455  143.404  4.715  5.954  49.762
 80.048  45.358 50.72  317.268 100.893  451.518  408.587 44.764  249.583  7.818  7.714  70.504
103.861  65.349 57.831 386.17  116.261  556.867  519.888 47.611  455.51  20.338 10.865 100.243
158.749  77.315 60.758 430.178 193.246 1025.525  997.986 76.465 1052.9   46.834 15.954 173.844
180.309  99.018 65.762 497.751 212.93  1131.788 1169.84  91.758 1475.856 76.262 25.786 189.405
199.36  102.155 70.544 534.113 247.517 1379.818 1381.824 96.78  1850.244 99.602 27.293 205.981
```

```
⍝ Co-dfns CPU vs. Chez Scheme Nanopass
0.2   0.5   0.5    1     0.5     1       1     1      0.5   1     1      1
1.5   0.333 0.5    2     0.5     4       2     1      2     1     1      3
2     1.5   1      4     0.25    4       4     1.5    4     2     1      1
1.867 0.6   2      4     0.2     6       8     1      2.5   1     4      3
2.621 0.875 2.5   12     0.889   9      18     2.333  6     2     1      3
2.722 0.875 1.8   20     0.875  32      16     1.273 27     1     0.5   13
3.336 2.658 2.444  6.5   0.935  68      16.5   1.45  11.4   2     6      7.429
5.352 3.6   4.556 21.5   1.607  53      55.5   2.65  21.125 1     8     13
6.982 5.043 5.467 23     1.958 127      63.75  3.52  35.545 5     2.667 17.429
8.122 5.932 7.081 24.45  2.374  82      98     3.969 42.905 2.25  3     18.783
8.481 7.18  6.383 22.167 2.558  73.111  99.154 3.645 49.368 2.571 4.25  18.42
9.112 7.709 6.628 27.709 2.907  74.025  98.172 3.694 59.913 4.067 5     19.659
9.259 7.909 6.431 24.7   3.056  75.84   94.813 3.869 56.51  5.793 5.184 18.374
9.233 7.463 7.284 31.651 3.295  81.841 105.19  4.288 47.769 0.86  5.1   14.106
9.417 8.229 7.505 22.03  3.116  84.483 109.298 4.419 40.148 7.408 6.426 12.901

⍝ Co-dfns GPU vs. Chez Scheme Nanopass
 0.008  0.02   0.043   0.132  0.05    0.179   0.144  0.065   0.083  0.122  0.331  0.018
 0.06   0.017  0.036   0.243  0.045   0.61    0.245  0.058   0.153  0.111  0.282  0.093
 0.123  0.044  0.075   0.421  0.041   0.572   0.473  0.347   0.587  0.227  0.281  0.059
 0.163  0.039  0.071   0.459  0.091   0.96    1.006  0.079   0.747  0.114  1.042  0.18
 0.399  0.083  0.293   1.286  0.658   2.494   1.992  0.685   1.697  0.204  0.262  0.301
 0.72   0.154  0.555   2.141  1.149   4.142   3.853  1.036   2.147  0.114  0.53   0.655
 1.747  1.011  1.155   4.862  2.161   8.802   7.178  2.041   5.54   0.215  1.526  1.176
 4.658  2.265  3.547  15.493  6.102  23.966  21.741  5.291  16.121  0.412  1.774  3.22
 9.786  5.237  5.549  32.184 12.222  52.081  46.56   8.476  28.402  1.018  2.972  7.456
17.351  9.302 10.02   58.457 19.479  91.682  86.345 12.334  54.107  1.414  3.883 14.574
24.469 14.051 12.435  85.853 26.369 137.165 126.816 14.384  97.892  2.199  5.581 21.849
35.034 21.419 16.727 115.102 34.537 188.941 177.771 16.921 196.256  7.902 10.865 34.141
42.352 28.745 18.183 135.587 40.091 225.743 224.646 18.873 300.289 16.921 13.846 42.272
50.566 32.251 22.604 173.842 47.658 283.163 293.268 22.891 451.143 29.123 16.062 49.992
56.157 39.203 24.904 193.566 53.852 315.706 329.702 24.194 544.707 37.091 20.256 52.185
```

APPENDIX G. RAW TIMINGS (SECONDS)

```
⍝ Racket Nanopass
  0.0106  0.0016  0.0008  0.0012  0.0014  0.0014  0.0004  0.0012  0.0014 0       0        0.0012
  0.0122  0.002   0.001   0.001   0.001   0.002   0.001   0.001   0.001  0       0        0.0022
  0.0236  0.004   0.0016  0.0024  0.0024  0.0024  0.0024  0.0022  0.0026 0.0004  0        0.0032
  0.0458  0.0078  0.0038  0.0044  0.0052  0.0048  0.0042  0.0042  0.0048 0.0002  0        0.0064
  0.0924  0.0156  0.0078  0.0082  0.0106  0.0094  0.0088  0.0086  0.01   0.0002  0.0002   0.013
  0.1936  0.0412  0.0152  0.0162  0.0202  0.0186  0.0172  0.0174  0.0198 0.0002  0.0006   0.027
  0.4716  0.0856  0.0394  0.0434  0.0502  0.0468  0.044   0.0454  0.051  0.001   0.0006   0.0642
  1.0226  0.1712  0.089   0.0956  0.1104  0.1024  0.0974  0.1008  0.1142 0.0014  0.0012   0.139
  2.1168  0.3548  0.1808  0.192   0.2312  0.2136  0.2038  0.2026  0.2398 0.0032  0.0022   0.2928
  4.5368  0.971   0.3598  0.3934  0.4738  0.4314  0.4074  0.4218  0.4776 0.006   0.0046   0.59
  8.9558  1.7754  0.7342  0.7864  0.9512  0.8664  0.8306  0.844   0.9566 0.0128  0.0094   1.1888
 17.9096  3.4244  1.4894  1.599   1.9026  1.7454  1.6652  1.713   1.919  0.0314  0.019    2.4012
 47.629   6.6542  2.9438  3.1816  5.7534  5.5814  5.3914  5.1284  5.9046 0.093   0.0454   6.9518
 98.172  15.499   5.8698  6.3438 11.6406 10.7294 10.574  11.2712 11.9078 0.2252  0.131   13.9802
208.4924 30.7262 11.8024 12.7658 25.144  24.4438 23.3622 23.4544 25.8834 0.5172  0.2684  30.124

⍝ Chez Scheme Nanopass
 0.0002  0.0002 0       0       0       0       0       0       0       0       0
 0.0018  0.0002 0       0.0002  0       0.0004  0.0002  0       0.0002  0       0       0.0006
 0.004   0.0006 0.0002  0.0004  0       0.0004  0.0004  0.0006  0.0008  0.0002  0       0.0004
 0.0056  0.0006 0.0002  0.0004  0.0002  0.0006  0.0008  0.0002  0.001   0       0.0004  0.0012
 0.0152  0.0014 0.001   0.0012  0.0016  0.0018  0.0018  0.0014  0.0024  0.0002  0       0.0024
 0.0294  0.0028 0.0018  0.002   0.0028  0.0032  0.0032  0.0028  0.0054  0       0.0002  0.0052
 0.0774  0.0202 0.0044  0.0052  0.0058  0.0068  0.0066  0.0058  0.0114  0.0002  0.0006  0.0104
 0.2344  0.0504 0.0164  0.0172  0.0196  0.0212  0.0222  0.0212  0.0338  0.0004  0.0008  0.0312
 0.5474  0.118  0.0328  0.0414  0.0466  0.0508  0.051   0.0528  0.0782  0.001   0.0016  0.0732
 1.2606  0.2432 0.0878  0.0978  0.113   0.1148  0.1176  0.1286  0.1802  0.0018  0.003   0.1728
 2.7376  0.55   0.18    0.2128  0.2486  0.2632  0.2578  0.2712  0.3752  0.0036  0.0068  0.3684
 6.0412  1.1224 0.4308  0.4766  0.5652  0.5922  0.5694  0.6088  0.8268  0.0122  0.019   0.8178
12.7068  2.474  0.881   1.0028  1.1936  1.2286  1.2136  1.2658  1.684   0.0336  0.0394  1.6904
27.5314  5.0482 2.0176  2.2156  2.6054  2.6844  2.6508  2.8118  3.64    0.086   0.0816  3.69
58.7294 11.7916 4.1666  4.6264  5.4706  5.5928  5.5742  5.8634  7.62    0.1926  0.1992  7.632

⍝ Co-dfns CPU
0.001  0.0004 0.0002 0       0.0002 0       0       0       0.0002 0       0       0
0.0012 0.0006 0.0002 0       0.0002 0       0       0       0       0       0       0.0002
0.002  0.0004 0.0002 0       0.0004 0       0       0.0004  0.0002 0       0       0.0004
0.003  0.001  0       0       0.001  0       0       0.0002  0.0004 0       0       0.0004
0.0058 0.0016 0.0004 0       0.0018 0.0002  0       0.0006  0.0004 0       0       0.0008
0.0108 0.0032 0.001  0       0.0032 0       0.0002  0.0022  0.0002 0       0.0004  0.0004
0.0232 0.0076 0.0018 0.0008  0.0062 0       0.0004  0.004   0.001  0       0       0.0014
0.0438 0.014  0.0036 0.0008  0.0122 0.0004  0.0004  0.008   0.0016 0.0004  0       0.0024
0.0784 0.0234 0.006  0.0018  0.0238 0.0004  0.0008  0.015   0.0022 0.0002  0.0006  0.0042
0.1552 0.041  0.0124 0.004   0.0476 0.0014  0.0012  0.0324  0.0042 0.0008  0.001   0.0092
0.3228 0.0766 0.0282 0.0096  0.0972 0.0036  0.0026  0.0744  0.0076 0.0014  0.0016  0.02
0.663  0.1456 0.065  0.0172  0.1944 0.008   0.0058  0.1648  0.0138 0.003   0.0038  0.0416
1.3724 0.3128 0.137  0.0406  0.3906 0.0162  0.0128  0.3272  0.0298 0.0058  0.0076  0.092
2.9818 0.6764 0.277  0.07    0.7908 0.0328  0.0252  0.6558  0.0762 0.1     0.016   0.2616
6.2362 1.433  0.5552 0.21    1.7556 0.0662  0.051   1.3268  0.1898 0.026   0.031   0.5916
```

```
⍝ Co-dfns GPU
0.0254 0.0098 0.0023 0.0008 0.002  0.0006 0.0007 0.0015 0.0012 0.0008 0.0003 0.0054
0.0302 0.0121 0.0028 0.0008 0.0022 0.0007 0.0008 0.0017 0.0013 0.0009 0.0004 0.0065
0.0325 0.0138 0.0027 0.0009 0.0025 0.0007 0.0008 0.0017 0.0014 0.0009 0.0004 0.0068
0.0344 0.0153 0.0028 0.0009 0.0022 0.0006 0.0008 0.0025 0.0013 0.0009 0.0004 0.0067
0.0381 0.0169 0.0034 0.0009 0.0024 0.0007 0.0009 0.002  0.0014 0.001  0.0004 0.008
0.0408 0.0182 0.0032 0.0009 0.0024 0.0008 0.0008 0.0027 0.0025 0.0009 0.0004 0.0079
0.0443 0.02   0.0038 0.0011 0.0027 0.0008 0.0009 0.0028 0.0021 0.0009 0.0004 0.0088
0.0503 0.0222 0.0046 0.0011 0.0032 0.0009 0.001  0.004  0.0021 0.001  0.0005 0.0097
0.0559 0.0225 0.0059 0.0013 0.0038 0.001  0.0011 0.0062 0.0028 0.001  0.0005 0.0098
0.0727 0.0261 0.0088 0.0017 0.0058 0.0013 0.0014 0.0104 0.0033 0.0013 0.0008 0.0119
0.1119 0.0391 0.0145 0.0025 0.0094 0.0019 0.002  0.0189 0.0038 0.0016 0.0012 0.0169
0.1724 0.0524 0.0258 0.0041 0.0164 0.0031 0.0032 0.036  0.0042 0.0015 0.0017 0.024
0.3    0.0861 0.0485 0.0074 0.0298 0.0054 0.0054 0.0671 0.0056 0.002  0.0028 0.04
0.5445 0.1565 0.0893 0.0127 0.0547 0.0095 0.009  0.1228 0.0081 0.003  0.0051 0.0738
1.0458 0.3008 0.1673 0.0239 0.1016 0.0177 0.0169 0.2423 0.014  0.0052 0.0098 0.1462
```

APPENDIX H. DISPLAYING TREES

```
⍝ dct :: Connectors (Get dlk Join) Formatted_Children
⍝        -> Connected_Children
⍝ Joins connectors on a rendered child tree along edge given by Get
dct←{⍺[(2×2≠/n,0)+(1↑⍨≠⍵)+m+n←⍵⍳\⍵m←' '≠⍺⍺ ⍵]⍵⍵ ⍵}

⍝ dlk :: Parent (Connector dlk Axis) Connected_Children -> Subtree
⍝ Links a parent with its prepared children using Connector along Axis
dlk←{((x⌷⍴⍵)↑[x←2|1+⍵⍵]⍺),[⍵⍵]⍺⍺⍀(⊂0 0)⍤('┌'=⊃⍵)⊢⍵}

⍝ dwh :: Children Parent -> Subtree
⍝ Horizontally render a parent and its children
dwh←{⍵('┬'dlk 1)' │├┌└─'(0⌷⍉)dct,⊃⍪/((≠¨⍺),¨⊂⌈/≠∘⍉¨⍺)↑¨⍺}

⍝ dwv :: Children Parent -> Subtree
⍝ Vertically render a parent and its children
dwv←{k←⊃{⍺,' ',⍵}/(1+⌈/≠¨⍺){⍺↑⍵⍪⍨⍤'│'↑⍨≠⍉⍵}¨⍺
 ⍵('├'dlk 0)' ─┬┌┐│'(0⌷⊢)dct(⍪⍨1↓⊢)k}

⍝ pp3 :: [Labels] (draw pp3) Parent -> Tree_Diagram
⍝ Pretty print a tree given node labels and a parent vector using draw
pp3←{⍺←'○' ⋄ d←(⍳≠p)≠p←⍵ ⋄ _←⍵{z⍪d+←⍵≠z←⍺[⍵]}⍤≡⍵
 lyr←{i←⍸⍺=d ⋄ k v←↓⍉p[i],∘⊂⌸i ⋄ (⍵∘{⍺[⍵]}¨v)⍺⍺¨@k⊢⍵}
 (p=⍳≠p)⌿⊃⍺⍺ lyr⌿(1+⍳⌈/d),⊂⍉∘⍪∘⍋¨(≠⍵)⍴⍺}
```

REFERENCES

Bernecky, R. 1997. *Apex: The APL parallel executor.*

—. 1999. Reducing computational complexity with array predicates. *ACM SIGAPL APL Quote Quad* 29 (3): 39–43.

—. 2003. An SPMD/SIMD parallel tokenizer for APL. In *Proceedings of the 2003 conference on APL: Stretching the mind.* 21–32. ACM.

Bernecky, R. and S. B. Scholz. 2015. Abstract expressionism for parallel performance. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming,* 54–59. New York: ACM.

Budd, T. A. 1984. An APL compiler for a vector processor. *ACM Transactions on Programming Languages and Systems* 6 (3): 297–313. ACM.

—. 2012. *An APL compiler.* Springer Science & Business.

Bunda, J. and J. A. Gerth. 1984. APL two by two: Syntax analysis by pairwise reduction. *ACM SIGAPL APL Quote Quad* 14: 85–94. ACM.

Chaudhuri, S. 1998. An overview of query optimization in relational systems. In *Proceedings of the 17th ACM Symposium on Principles of Database Systems,* 34–43. ACM.

Ching, W. M. 1990. Automatic parallelization of APL-style programs. *ACM SIGAPL APL Quote Quad* 20: 76–80. ACM.

Ching, W. M. and A. Katz. 1994. An experimental APL compiler for a distributed memory parallel machine. In *Proceedings of the 1994 ACM/IEEE conference on supercomputing,* 59–68. IEEE Computer Society Press.

Ching, W. M., P. Carini, and D. C. Ju. 1993. A primitive-based strategy for producing efficient code for very high level programs. *Computer Languages* 19 (1): 41 – 50.

Cong, G., S. Kodali, S. Krishnamoorthy, D. Lea, V. Saraswat, T. Wen. 2008. Solving large, irregular graph problems using adaptive work-stealing. In *37th international conference on parallel processing* (September). IEEE.

Dyalog. 2019a. Scholisms. http://johnscholes.rip/scholisms/

Dyalog. 2019b. Searchable Dyalog, FinnAPL, and APL2 idiom lists. https://miserver.dyalog.com/Examples/Applications/Idiom_Search.mipage

Eisenberg, M. and H. A. Peelle. 1987. APL thinking: examples. *ACM SIGAPL APL Quote Quad* 17 (4): 433–440. New York: ACM.

Falkoff, A. and K. E. Iverson. 1978. The evolution of APL. *ACM SIGPLAN Notices* 13 (8): 47–57. New York: ACM.

Gibbons, J. 2017. APLicative programming with naperian functors. *Programming Languages and Systems ESOP 2017: Lecture Notes in Computer Science* 10201:556–583. Berlin: Springer.

Goldfarb, M., Y. Jo, and M. Kulkarni. 2013. General transformations for GPU execution of tree traversals. In *SC '13 Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (November): 10.1–10.12. New York: ACM.

Hallenberg, N. 2016. The use of APL in SimCorp Dimension. Presented at *Dyalog '16* (Glasgow). https://www.dyalog.com/uploads/conference/dyalog16/presentations/L06_Use_of_APL_in_SimCorp.pdf

Harris, M., S. Sengupta, and J D. Owens. 2007. Parallel prefix sum (scan) with CUDA. *GPU Gems* 3 (39): 851–876. Nvidia.

Henglein, F. and R. Hinze. 2013. Sorting and searching by distribution: From generic discrimination to generic tries. *Programming Languages and Systems* (December): 315–332. Springer.

Hors, A. L., P. L. Hegaret, L. Wood, G. Nicol, J. Robie, M. Champion, S. Byrne, ed. 2004. *Document object model (DOM) level 3 core specification.* W3C Recommendation April 2004. W3C. http://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407

Hsu, A. W. 2014. Co-dfns: Ancient language, modern compiler. In *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, 62. ACM.

—. 2015. Accelerating information experts through compiler design. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, 37–42. ACM.

—. 2016. The key to a data parallel compiler. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, 32–40. New York: ACM.

—. 2017. Design patterns vs. anti-pattern in APL. Presented at *FunctionalConf 2017*. Bengaluru: FnConf.

Hui, R. K. 1995. Rank and uniformity. *ACM SIGAPL APL Quote Quad* 25:83–90. ACM.

—. 2007. Essays/Key. http://www.jsoftware.com/jwiki/Essays/Key.

Iverson, K. E. 1962. *A programming language*. New York: John Wiley & Sons.

—. 1983. Rationalized APL. IP Sharp Associates.

—. 2007. Notation as a tool of thought. *ACM SIGAPL APL Quote Quad* 35 (1–2): 2–31. New York: ACM.

Jo, Y., M. Goldfarb, and M. Kulkarni. 2013. Automatic vectorization of tree traversals. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques* (September). IEEE.

Ju, D. C. and W. M. Ching. 1991. Exploitation of APL data parallelism on a shared-memory MIMD machine. *ACM SIGPLAN Notices* 26:61–72. ACM.

Ju, D. C., W. M. Ching, and C. L. Wu. 1991. On performance and space usage improvements for parallelized compiled APL code. *ACM SIGAPL APL Quote Quad* 21 (4): 234–243. ACM.

Keep, A. W. and R. K. Dybvig. 2013. A Nanopass framework for commercial compiler development. *ACM SIGPLAN Notices* 48:343–350. ACM.

Kepner, J. and J. Gilbert. 2011. *Graph algorithms in the language of linear algebra*. Philadelphia: Soc. for Industrial and Applied Math.

Kromberg, M. 2018. Workshop conducted at *Functional Conf 2018*. Bengaluru, India.

Knuth, D. and D. Doernberg. 1993. Computer Literacy Bookshops Interview with Donald Knuth. By Dan Doernberg. http://tex.loria.fr/historique/interviews/knuth-clb1993.html .

Lochbaum, M. 2018a. The interpretive advantage. Presented at the 2018 Dyalog User Meeting. Belfast: Dyalog. https://www.dyalog.com/user-meetings/dyalog18.htm

—. 2018b. Sub-nanosecond Searches Using Vector Instructions. Presented at the 2018 Dyalog User Meeting. Belfast: Dyalog. https://www.dyalog.com/user-meetings/dyalog18.htm

Liu, J., N. Hegde, M. Kulkarni. 2016. Hybrid CPU-GPU scheduling and execution of tree traversals. In *Proceedings of the 2016 International Conference on Supercomputing* (June). New York: ACM.

Malcom, J., P. Yalamanchili, C. McClanahan, V. Venugopalakrishnan, K. Patel, J. Melonakos. 2012. ArrayFire: A GPU acceleration platform. In *Proceedings of SPIE* 8403 (May). SPIE.

Marlow, S. and S. P. Jones. 2012. The Glasgow Haskell compiler. In *The Architecture of Open Source Applications* 2. http://www.aosabook.org/en/ghc.html.

McIntyre, D. B. 1991. Language as an intellectual tool: From hieroglyphics to APL. *IBM Systems Journal* 30 (4): 554–581. IBM.

McKenney, P. E., ed. 2017. *Is parallel programming hard, and, if so, what can you do about it?* arXiv. https://arxiv.org/pdf/1701.00854.pdf .

Mendez-Lojo, M., M. Burtscher, and K. Pingali. 2012. A GPU implementation of inclusion-based points-to analysis. *ACM SIGPLAN Notices* 47 (8): 107–116. ACM.

Metzger, R. C. 1981. APL thinking finding array-oriented solutions. *ACM SIGAPL APL Quote Quad* 12 (1): 212–218. New York: ACM.

Mullin, L. M. R. 1988. A mathematics of arrays. PhD diss., Univ. at Albany.

Murphy, R., K. B. Wheeler, B. W. Barrett, and J. A. Ang. 2010. Introducing the Graph 500. *Cray Users Group (CUG)* 19 (May): 45–74.

Perlis, A. J. 1977. In praise of APL: A language for lyrical programming. *ACM SIGAPL APL Quote Quad* 8 (2): 44–47. New York: ACM.

Pesch, R. H. 1982. Book review: FinnAPL idiom library and FinnAPL pocket idiom library. *ACM SIGAPL APL Quote Quad* 13 (2): 26–27. New York: ACM.

Prabhu, T., S. Ramalingam, M. Might, and M. Hall. 2011. EigenCFA: Accelerating flow analysis with GPUs. *ACM SIGPLAN Notices* 46 (1): 511–522. ACM.

Ren, B., G. Agrawal, J. R. Larus, T. Mytkowicz, T. Poutanen, W. Schulte. 2012. Fine-grained parallel traversals of irregular data structures. In *PACT '12: Proceedings of the 21st international conference on parallel architectures and compilation techniques* (September): 461–462. New York: ACM.

Satish, N., M. Harris, and M. Garland. 2009. Designing efficient sorting algorithms for manycore GPUs. In *IEEE international symposium on parallel & distributed processing* (May). IEEE.

Saxonica, M. K. 2017. *XSL transformations (XSLT) version 3.0.* W3C Recommendation (June 2017). W3C. https://www.w3.org/TR/2017/REC-xslt-30-20170608/ .

Scholes, J. 2014. Depth-first search in APL. In Dyalog's dfns library. http://dfns.dyalog.com/n_trav.htm

Schwarz, W. 1991. Acorn run-time system for the CM-2. *Arrays, Functional Languages, and Parallel Systems*, 35–57. Springer.

Sengupta, S., M. Harris, Y. Zhang, and J. D. Owens. 2007. Scan primitives for GPU computing. *Graphics Hardware* 2007:97–106.

Silberstein, M., A. Schuster, D. Geiger, A. Patney, J. D. Owens. 2008. Efficient computation of sum-products on GPUs through software-managed cache. In *ICS '08 Proceedings of the 22nd annual international conference on supercomputing* (June): 309–318. New York: ACM.

Slepak, J., O. Shivers, and P. Manolios. 2014. An array-oriented language with static rank polymorphism. *Programming Languages and Systems LNCS* 8410: 27.

Taylor, S. 2005. Pair programming with users. *Vector* 22 (1). London: British APL Association.

Whitney, A. 2009. A conversation with Arthur Whitney. By Bryan Cantrill. *ACM Queue* 7 (2): 12–19. New York: ACM.

Zhang, F., Di P., Zhou H., Liao X., and Xue J. 2016. RegTT: Accelerating tree traversals on GPUs by exploiting regularities. In *45th International Conference on Parallel Processing* (August). IEEE.

CURRICULUM VITAE

Education

Ph.D., Computer Science	2019	Indiana University, Bloomington, IN
M.S., Computer Science	2010	Indiana University, Bloomington, IN
B.S., Computer Science	2008	University of Missouri, St. Louis, MO

Work Summary

(Dyalog, Indiana University) Co-dfns Compiler for APL.

(Indiana University) Foundations in Science and Mathematics computer science curriculum

(Contract) Virtual Inventory calculation engine for textiles manufacturing

(Indiana University) Compiler framework for undergraduate compilers course

(Indiana University) MAGS grading system for introductory computing courses

Academic Honors

Summa Cum Lauda, University of Missouri, St. Louis, MO

Student Marshal, University of Missouri, St. Louis, MO

Software Development

ChezWEB system of hygienic literate programming

APL implemented line editor

DocBook SIGPLAN template for ACM publications

Mystika encryption library

Chez port of the SRFIs library

E-Commerce web sites for small business expansions

Chez Scheme Sockets implementation

Chez Scheme Gopher Server, Goscher

Maintained and updated the NN newsgroup client

Maintained the Scheme Working Groups issue tracker and engine

Publications

Hsu, Aaron W. "The key to a data parallel compiler." In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, pp. 32-40. ACM, 2016.

Hsu, Aaron W. "Accelerating information experts through compiler design." In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, pp. 37-42. ACM, 2015.

Hsu, Aaron W. "Co-dfns: Ancient language, modern compiler." In *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, p. 62. ACM, 2014.

Hsu, Aaron W. "Hygienic Literate Programming: Lessons from ChezWEB." 2011.

Hsu, Aaron W., and Claire Alvis. "Learning from GOATS." *Journal of Computing Sciences in Colleges* 27, no. 1 (2011): 35-35.

Hsu, Aaron, and Geoffrey Brown. "Dependency Analysis of Legacy Digital Materials to Support Emulation Based Preservation." *IJDC* 6, no. 1 (2011): 99-110.

HSU, AARON W., ALARIC SNELL, ARTHUR A. GLECKLER, BENJAMIN L. RUSSEL, BRADLEY LUCIER, DAVID RUSH, EMMANUEL MEDERNACH, and JEFFREY T. READ. "Revised7Report on the Algorithmic Language Scheme." (2011).

Hsu, Aaron W. "Implementing User-level Value-weak Hashtables." In *2010 Workshop on Scheme and Functional Programming*, p. 15.

Hsu, Aaron W. "Descot: Distributed Code Repository Framework." *Technical Report CPSLO-CSC-09-03*: 86.

Hsu, Aaron W., and Michel Salim. "Formal Verification of Scheme Module and Library Transformations." (2009).

Presentations/Talks/Workshops

BOB 2019. The Way of APL. Berlin, Germany.

Dyalog 2019. Lessons for the Masses from the Trenches of Co-dfns.

Functional Conf. 2018. Array-oriented Functional Programming. Bengaluru, India.

Functional Conf. 2018. Does APL Need a Type System? Bengaluru, India.

Dyalog 2018. Co-dfns 2018 – What's New?

Dyalog 2018. High-performance Tree Wrangling, the APL Way.

LambdaConf 2018. Parallel-by-construction Tree Manipulation with APL.

Functional Conf. 2017. APL Patterns vs. Anti-patterns. Bengaluru, India.

YouTube Livestream. 2017. Co-dfns Compiler Architecture and Design.

Dyalog 2017. Patterns and Anti-patterns in APL: Escaping the Beginner's Plateau.

Dyalog 2017. Co-dfns Report 2017: Ease of Use, Reliability and Features.

Dyalog 2017. Co-dfns Compiler: Hands-on GPU programming with APL.

LambdaConf 2017. Functional Array Funhouse Intensive.

Dyalog 2016. Compiling ANN & other APL Code.

Dyalog 2016. Co-dfns Report: GPU Performance, Workflow, and Usability.

Dyalog 2015. Using Co-dfns to Accelerate APL Code.

Dyalog 2014. Co-dfns Report: Performance and Reliability Prototyping.

Dyalog 2013. Co-dfns compiler.

Dyalog 2013. Computer Science Outreach and Education with APL.

Teaching

P423 Compilers Course Instructor for Indiana University.

B551 Elements of Artificial Intelligence Assistant Instructor for Indiana University.

C211 Introduction to Computing Assistant Instructor for Indiana University

P523 Compilers Assistant Instructor for Indiana University

Indiana University Foundations in Science and Mathematics Computer Science instructor