Array Morphology*

Robert Bernecky
Snake Island Research Inc
18 Fifth Street, Ward's Island
Toronto, Ontario M5J 2B9
Canada
+1 416 203 0854
bernecky@acm.org

Abstract

Array morphology is the study of the form, structure, and evolution of arrays. An *array annotation* for a program written in an applicative array language is an abstract syntax tree for the program, amended with information about the arrays created by that program. Array notations are useful in the production of efficient compiled code for applicative array programs. Array morphology is shown to be an effective compiler writer's tool. Examples of an array annotator in action are presented, showing its value in array morphology. Array morphology is shown to provide methods for static detection of certain classes of programming errors.

Assertions are a generalization of declarations that offer significant benefits to application writers as well as compiler writers. Although assertions are executable code, they can often be evaluated at compilation time. Assertions, and therefore declarations, may be represented as conjunctions, and are, therefore, conforming extensions to ISO Standard APL. A *domain conjunction* is offered as an example of how assertions might be defined in APL or J.

*This paper originally appeared in the ACM SIGAPL APL93 Conference Proceedings. [Ber93]

1 Introduction

This paper defines *array morphology* and presents some of the results obtained in the creation of two pieces of an APL compiler – the tokenizer/parser and the *array annotator*. The tokenizer creates an *abstract syntax tree* [ASU86], or *AST*, from the source code. The annotator amends the AST with morphological information about the form and structure of the arrays created by the execution of the program.

Annotation performs well for certain user-defined APL functions, and poorly for others. Although declarations can solve most of the problems, they are antithetical to the design of APL. The use of *assertions* as generalized declarations is presented as a more general solution to the problem.

2 Array Morphology

Array property analysis for applicative array languages has much in common with data flow analysis (DFA) [ASU86] for traditional scalar-oriented languages. However, such analysis differs from data flow analysis in enough respects that it merits a distinct name. Hence, the analysis of array properties will be denoted as *array morphology*, since it involves the study of the form, structure, and evolution of arrays, just as geomorphology is the study of the form, structure, and configuration of geologic structures.

Array morphology is a method for establishing and tracking array properties. It is particularly useful for the static determination of the type, rank, shape, and value of arrays arising from the execution of APL, J, and other array languages.

2.1 The Benefits of Array Morphology

Array morphology statically determines properties of the arrays resulting from the execution of a program written in an applicative array language. Knowledge of these properties has benefits for both interpreters and compilers, including:

- Elimination of run-time checking. If the argument types to a specific primitive are statically determined and are known to be compatible for the primitive, then run-time type checking for that primitive can be eliminated. Since type and conformability checking are responsible for much of the overhead of traditional APL systems, their removal is key to improving the performance of such systems. Similarly, knowledge of rank, shape, and value allow other run-time conformability checks to be eliminated.

- Static determination of *compute type*. The compute type of an operation depends on the argument types. Typically, the compute type is the more encompassing of the two types. For example, adding Booleans to integers uses an integer compute type, whereas adding complex numbers to floats uses a complex compute type. Static determination of compute type eliminates yet another form of run-time overhead.

- Reuse of arrays. Knowledge of array properties may permit them to be reused, thereby reducing the storage management services required by the application. Reference count maintenance may be simplified, by application of techniques such as that proposed by Skedzielewski and Simpson [SS88].

- Reduced descriptor management. Knowledge of array properties can reduce the amount of array descriptor management overhead in an executing program.

- Fault detection. Several families of common programming faults, or bugs, can be detected statically, without ever executing the program. This capability is similar to the *lint* facility in C.

- Identities. A number of useful identities can be detected at compile time, including many that are too expensive to determine at run-time in an interpreted environment.

- Arithmetic progression vectors. Some APL interpreters [Ive73] include support for *j-vectors* or *arithmetic progression vectors*. In their simplest form, these encode expressions of the form `x+y×⍳z` as the list `x,y,z`. Further operations using j-vectors as arguments are often able to be optimized, or to produce new j-vectors as results, with large savings in execution time and storage. J-vectors are, however, considered a curse by most implementors, as they increase run-time overhead and are a maintenance nightmare. Static detection and propagation of j-vectors offers the same benefits with no run-time overhead.

```
r←benchloop n
⍝ No optimization.
r←(⍳0)⍴⌊n ⍝ Note that r is not used.
n←2500×n
L:
→(0<n←n-1)⍴L
r←1
```

Figure 1: Benchloop without declarations

- Shape calculus. Tracking of object shapes and ranks may allow run-time code to be eliminated, by allowing conformability checks to be made symbolically at compile time. This has particular value when used in conjunction with partial properties.

3 Abstract Syntax Trees

Abstract syntax trees, or ASTs, ease the problems of analyzing and optimizing code undergoing compilation by parsing free-form source code to create a tabular representation of the same program in a form that is amenable to rapid analysis and modification. Figure 1 is the APL code for a simple program `benchloop`. The first four columns of Figure 5 are the abstract syntax tree created for `benchloop` by the tokenizer/parser.

3.1 AST Structure

The data structure used to represent abstract syntax trees is a six-column table of arrays. Each row of the table represents the computation performed by a single APL primitive verb, adverb, or conjunction. The format and content of each row of the table is:

Target The result of the computation of a single primitive. It is the name of a noun in the original program, or a temporary name generated by the compiler, such as `T_666`. All temporary arrays created by the parser are single assignment.

⍺ The left argument to the primitive, represented as the 0-origin index of the AST row that generated the argument. The formal arguments to the user-defined verb being compiled appear as dummy AST entries at the top of the AST. For monadic verbs, this element is elided.

Verb The verb, adverb, or conjunction associated with this row. In the case of derived verbs, it is the entire adverbial expression, such as `+.×` for matrix product, or `+/` for summation.

Axis The explicit rank [Ber87], if any, applied to the verb. The rank is, in a sense, a regular, generalized descendant to the irregular *bracket axis* notation of ISO Standard APL [Int84]. Since rank is not used in the examples presented here, it does not appear in the tables.

⍵ The right argument to the verb, represented similarly to the left argument.

Index The list of expressions that index the array argument, represented as a list of array names. If the verb under execution is an indexed reference or indexed assignment, the syntax of APL permits an arbitrary number of arguments to the verb. If the verb is not an indexing expression, this element is elided.

The annotator amends the AST with additional elements, which are described in Section 5.1.

3.2 AST Creation

The AST is created by a tokenizer/parser similar to that used in a direct code-generating compiler. The difference is that, instead of emitting machine instructions, or code in a target language, the parser emits rows of the AST. Names of temporary results are kept on a parser stack and removed at an appropriate time later in the parsing of the APL sentence.

The tokenizer operates in parallel on the entire source program at once, without any looping. The parser iterates over each token generated by the tokenizer.

4 Array Annotation

Once the AST has been constructed, it can be annotated using array morphology and data flow analysis. *Array annotation* is the static amendment of an AST with information about the morphological properties of the arrays created by the verbs of a program. Of particular interest to the compiler writer are the type, rank, shape, and value of such arrays. Other properties are also of interest, but have less immediate value.

Data flow analysis and array notation are related, but their main focus differs. Data flow analysis performs several duties, including:

- Segmenting the source program into *basic blocks*. Basic blocks are straight-line code with no entry or exit except at the top or bottom of a block. *Local analysis* is performed independently upon each basic block. *Global analysis* is performed across all basic blocks.

- Tracing the lifetimes of variables, as defined by their definitions and uses within and across basic blocks. In a traditional scalar compiler, these *def/use chains* are used for such purposes as register allocation and code hoisting.

Even though most APL programs are loop-free, having few basic blocks, data flow analysis is still required for morphological analysis. The current annotator assumes that any named argument has the properties of the *closest preceding definition* of that name in the AST. That is, if `d` is the list of row numbers within the AST where the name in question is defined, and `u` is the index of a use of that name, the closest preceding definition is `⌈/(d<u)/d`. As will be seen later on, this assumption can lead to difficulties, so even APL requires proper data flow analysis.

The annotator uses the AST to infer properties about the arrays created by the program being compiled, by examining each verb and adverb that can potentially be executed during the lifetime of an APL program. For example, it can deduce that the result of the index generator verb is always an integer list. This information may then be used later in the annotation process to determine the properties of a verb that takes the result of the index generator as an argument. This is denoted as *forward morphological analysis*.

The annotator may also be able to deduce properties of the arguments to a verb. For example, ISO Standard APL restricts the left argument of the indexof verb to be a list. This information can be used to deduce properties of the arguments of the earlier verb that created the left argument. This is denoted as *backward morphological analysis*.

4.1 Assumption of Correctness

Array morphology assumes that the program under analysis is correct. Deductions about array properties are based on the known characteristics of primitives and arrays executing in an error-free manner. Although the annotator can statically detect certain classes of error in programs, other types of source program errors will produce incorrect annotation, which may cause incorrect results of other run-time failure at unexpected locations in the program. This has obvious implications for programs which depend on event trapping for their execution.

5 How to Annotate an Array

This section describes the array annotator and shows its capabilities and limitations for each class of primitive it handles.

5.1 Annotated AST Structure

An annotated AST includes amendments made by morphological analysis. These amendments are represented in the AST as columns containing:

Class The class represented by the array created by each row of the AST. Its primary use is in propagation of constants, which is useful in partial evaluation. The class is represented as a character constant, such as `'c'` for constant, `'v'` for variable, etc.

Type The result type of each entry. This is most significant from the standpoint of compiled code, because type is the major factor controlling code generation. Type is maintained as one of the following character constants:

- *b* Boolean. One bit per element.
- *i* Integer. Typically 32-bit or 64-bit integer.
- *f* Float. Typically double precision IEEE floating point.
- *z* Complex. Typically a pair of floats.
- *c* Character. Typically 8-bit, 16-bit, or 32-bit bytes.
- *a* Array. Typically 32-bit or 64-bit pointers.

Rank The result rank of each entry. This is represented as a scalar between 0 and the maximum allowable rank array in the system.

Shape The result shape of each entry. This is represented as an integer list of zero or more elements.

Value The result value of each entry. This value is usually only known for constants and entries whose value is determined by constant propagation.

Unknown entries are represented as empty lists.

Figure 5 shows two abstract syntax trees for `benchloop` after the array annotator has processed it. One represents the original program as shown in Figure 1; the other reflects the program in Figure 4, after the introduction of declarations. Since the axis and index columns are empty in this example, they have been elided from the figure.

5.2 Annotation Methodology

During array annotation, information about the arguments to each verb is analyzed on a verb-dependent or derived-verb-dependent basis. Appropriate entries are made in the AST as information about type, rank, shape, and value is deduced. Tentative entries are never made – once an AST annotation entry is filled in, it is never changed.

This lack of tentative behavior is required because of the dependency of annotation on earlier annotation decisions. If an early decision turns out to be wrong, it may invalidate an arbitrary number of later annotation decisions. Therefore, the annotator takes a conservative approach and only fills in entries when there is no doubt as to their content. As will be shown later on, this isn't quite true, but it is quite effective.

This *single-assignment* approach permits the entire AST to be processed in parallel, at least conceptually. In the current annotator, each family of verbs (e.g, all dyadic scalar verbs (plus, minus, times, divide, maximum, nand, nor; all reductions, etc.) are analyzed in parallel. Iteration over the AST continues until no further entries can be filled in.

Except as noted below, the annotator does not currently propagate value information or perform partial evaluation based on known values.

The remainder of this section describes how each family of related APL primitives are handled. Most of the relevant properties are shown in Figure 2. For simplicity in presentation, some of the properties shown in the figure are approximations.

Indexed Reference APL's anomalous syntax for indexing permits the rank of the argument array to be deduced statically. For example, indexing of a list in APL is written as `x[i]` and indexing of a table is written as `x[i;j]`.

The rank of the result is deducible if the indexing arguments have known ranks. The result rank is the sum of the ranks of all the indexing arguments. In the case of elided indexing arguments, to select entire rows or columns, as in `x[i;]` to pick row(s) `i` from `x`, the rank of the result increases by one for each elided index.

The shape of the result is the catenation of the shapes of the indexing arguments, except when an indexing argument has been elided. In that case, the element of the result shape that corresponds to the elided index is the corresponding member of the shape of the array being indexed. It is only when that shape is also known that the result shape can be fully determined.

Indexed Assignment Indexed assignment can change the type of the array being assigned into. Therefore, indexed assignment data flow analysis and array morphology are conservative in their treatment of type.

Indexof Indexof verifies that its left argument is a list or is of unknown rank, and establishes a rank of 1 (list) for the left argument if its rank was previously unknown.

Name	Verb	Result type	Result rank	Result shape
Conjugate	+	T ω	ρρω	ρω
Negation	−	T ω	ρρω	ρω
Signum	×	I	ρρω	ρω
Reciprocal	÷	F	ρρω	ρω
Power of *e*	⋆	F	ρρω	ρω
Ceiling	⌈	I	ρρω	ρω
Floor	⌊	I	ρρω	ρω
Magnitude	|	T ω	ρρω	ρω
Logical not	~	B	ρρω	ρω
π times	○	F	ρρω	ρω
log_e	⍟	F	*	*
Add	+	α MT ω	*	*
Subtract	−	α MT ω	*	*
Multiply	×	α MT ω	*	*
Divide	÷	F	*	*
Power	⋆	α MT ω	*	*
Logarithm	⍟	F	*	*
Maximum	⌈	α MT ω	*	*
Minimum	⌊	α MT ω	*	*
Residue	|	α MT ω	*	*
Less than	<	B	*	*
Not greater	≤	B	*	*
Equal	=	B	*	*
Not less	≥	B	*	*
Not equal	≠	B	*	*
Or	∨	B	*	*
And	∧	B	*	*
Nor	⍱	B	*	*
Nand	⍲	B	*	*
IndexR	ω[i]	T ω	*	*
IndexA	ω[i]←	α MT ω	ρρω	ρω
Upgrade	⍋ω	I	1	1↑ρω
Downgrade	⍒ω	I	1	1↑ρω
Shape	ρω	I	1	ρρω
Reshape	αρω	T ω	ρ,α	,α
Assign	←	T ω	ρρω	ρω
Rotate	αφω	T ω	ρρω	ρω
Reversal	φω	T ω	ρρω	ρω
Rotate	α⊖ω	T ω	ρρω	ρω
Reversal	⊖ω	T ω	ρρω	ρω
Transpose	⍉ω	T ω	ρρω	φρω
Transpose	α⍉ω	T ω	*	*
Take	α↑ω	T ω	*	α
Drop	α↓ω	T ω	*	*
Set	α∊ω	B	ρρα	ρα
Indexof	α⍳ω	I	ρρω	ρω
Integers	⍳ω	I	1	ω
Catenate	α,ω	α MT ω	*	*
Replicate	α/ω	T ω	1⌈ρρω	*

* – too complicated for table
T – type
MT – maximum of types

Figure 2: Morphological properties of verbs

Index Generator If the argument value is known, the result shape is established.

Catenate The result shape is more complicated, but is roughly the shape of one argument except along the axis of catenation, where it is the sum of the shapes of both arrays along that axis. Treatment of scalars and degenerate arrays complicates matters considerably.

Reduce There are too many cases of reduction to enumerate here. Commonly used reductions have a Boolean (all, any) result or a result type which is the same, ignoring blowup, as the argument (summation, maximum). The result rank is `0⌈¯1+⍴⍴⍵`. The result shape is the argument shape with one element removed, depending on the axis of reduction.

Inner Product The inner product conjunction has an immense number of cases, again too many to present here. The result rank is `0⌈¯2+(⍴⍴⍺)+⍴⍴⍵`. The result shape is the catenation of the two argument shapes, with the trailing element of the left argument shape and the leading element of the right argument shape elided. Type analysis is fairly complicated, being based on both verb arguments to the conjunction as well as both array arguments to the derived verb. Nonetheless, it can be done, and the resulting code to support it is surprisingly short.

Outer Product Outer product is handled similarly to dyadic scalar verbs, except that outer product pairs each element of one argument with the entire other argument. Thus, the result type is identical to that of dyadic scalar verbs. The result rank is the sum of the argument ranks, and the result shape is the catenation of the argument shapes.

5.3 Loops

The presence of loops in programs makes it possible that the set of definitions entering a basic block may conflict. Such a conflict may merely mean that generated code quality will suffer. It may also mean that the program has a bug in it. Consider the following code fragment and the basic block starting with label `lp:`

```
k←5
lp: k←k+1
→(k<1000)/lp
```

Upon entry from the first line, the variable `k` is an integer scalar. The value upon entry from the branch label, however, has unknown characteristics until the entire basic block containing the branch to `lp` has been analyzed. But it cannot be analyzed until the characteristics of `k` are known! How can this Catch-22 problem be dealt with?

Several methods come to mind:

Greedy Take the reaching definition from the previous basic block at face value and analyze. When the analysis is complete, compare the values entering each block. If they differ, declare an error in the program. In the example above, this approach would, by happenstance, work perfectly.

Merge Combine the morphological information from different paths, taking the minimum of their intersection. In the above example, this would never resolve the type information of `k`, because the "don't know yet" fields would take precedence over the known ones.

Tentative Assume the basic block inputs are known, in the same manner as the greedy method. See if things work out at the end. If not, back off to more conservative information and try again.

Traceback Keep information which allows each morphological deduction to be traced to its source. If the sources agree at their confluence, things are fine. If not, then the program is more complex than we expect. For example, a function is may be called at times with matrix arguments, and at other times with list arguments. The YAT approach of generating separate code for each instance of such programs makes sense here.

5.4 Oh, what a tangled web we weave!

Strong threads connect executing APL primitives – array properties resulting from one primitive may directly or indirectly affect the properties of many others. Inability to deduce a property for one primitive's result may have a domino effect, making it impossible to deduce a large number of properties of arrays that are dependent upon the first one. Just as pulling a single thread from a cloth may cause its destruction, so the failure to deduce a single array property may cause many further annotations to fail.

On the other hand, this same sensitivity suggests that an interactive compiler that is able to ask the user for assistance in determining properties of specific arrays may be of great utility. Such a compiler might suggest: "If you declare the type of `x` on line 23 of function `foo`, I can increase the deduction score for type and rank from 30% to 97%."

Presentation of array connectivity can be made as a spider web on a graphical user interface, and the primitives and arrays colored in a way related to the presence or lack of morphological information related to each of them. However, as tempting as such a presentation might be, it is of little use to the unsighted and color blind among us, so other presentation facilities are required as well.

The connectivity of morphological array information among primitives suggests that there may not be large differences among primitive families in terms of their impact on annotation. Rather, their physical position in the source program may be a more critical factor in measuring the impact of a missing array property.

In order to evaluate the relative impact of annotators for different families of primitives, annotation was performed on `ls`, Mike Jenkins' model of the `APL\360` domino function [Jen70], with each family disabled independently.

The results of this are shown in Figure 3. Not surprisingly, failure to propagate assignment information caused severe problems. Scalar verbs and reshape took a heavy toll, likely due to their popularity. Indexed reference and, to a lesser degree, indexed assign caused trouble as well. Indexing gives strong hints as to rank, which suggests that more regular languages, such as J, may have more problems with array morphology than APL does. See Section 12 for more information on this topic.

More study of real world applications is called for, because it is clear that examination of a single program is not an adequate measure of annotator sensitivity.

5.5 How to Annotate an Array, Really

Manual annotation of an array is a good way to understand how general annotation works. It is a relaxation algorithm much like solving a crossword puzzle or cross sums:

- Start with an AST for the program you want to annotate. Catenate as many columns on the AST as you have fields you want to annotate. The figures in this paper, for example, annotate type, rank, shape, and value.

Disabled Handler	Number of deduced items Type	Rank	Shape	Value
Total possible	174	176	174	174
All enabled, no declarations	99	152	15	15
All enabled, w/declarations	174	176	22	15
Assignment	49	51	1	13
Monadic scalars	88	143	15	15
Dyadic scalars	60	92	6	15
Indexed assign	109	156	22	15
Indexed reference	89	71	8	15
Grade	146	160	20	15
Rotate, reversal	172	163	20	15
Shape	113	161	13	15
Reshape	95	155	22	15
Transpose	170	167	15	15
Reduction	133	127	9	15
Take	113	170	15	15
Drop	126	130	20	15
Inner product	157	165	15	15
Outer product	126	130	20	15
Set membership	174	176	22	15
Indexof	162	160	16	15
Index generator	163	163	21	14
Catenate	101	143	10	15
Replicate	171	173	22	15

Figure 3: Effect on annotation of `ls` of selectively disabling annotation handlers.

- Delete, modify, or add family handler functions as required, to perform the specific annotation you desire.

- Use information from entries you have already filled in to deduce properties of missing entries. Fill in those entries. It does not matter if you miss an entry on one pass through all family handlers, as long as you get it on a later pass. It *is* important, however, to do some work on every pass, filling in at least one annotation entry.

- Keep going until the table stops changing.

Doing non-zero work on each iteration guarantees completeness, because the table entries must eventually fill. It also allows a certain amount of sloppiness to exist harmlessly in the annotation algorithms. Such behavior is characteristic of certain simple algorithms and of unsynchronized parallel computation. The number of iterations required to fully annotate an AST with the current algorithm is no worse than linear in the number of AST entries. This suggests that annotation performance is better if multiple entries are filled in on each pass.

6 The Impact of Declarations

Declarations can be of great value in array morphology. Even though this is saying that it is easy to get good exam scores if someone tells you the answers, it turns out that a few hints go a long way. As shown in the top of Figure 3, annotation of `ls` without declarations is able to deduce the type of only about half of the arrays created by the program. However, if the annotator is told that the arguments are floating tables, then the annotator score for type and rank deduction jumps to 100%, showing that morphological analysis of programs can benefit substantially from

```
r←benchloopb n
⍝ Enable optimization.
n←(⍳0)⍴⌊n ⍝ Make n an integer scalar.
n←2500×n
L:
→(0<n←n-1)⍴L
r←1
```

Figure 4: Benchloop with declarations

a few declarations. This confirms Budd's findings regarding interprocedural analysis. [Bud85]

As a simple example of declarations in action, consider the two versions of function `benchloop` in Figures 1 and 4, and its two annotations in Figure 5.

The annotation of `benchloop` without declarations was unable to deduce information because it had no starting point. Adding the sentence `n←(⍳0)⍴⌊n` has the effect of making `n` an integer scalar after its execution. This gave the annotator a starting point, thereby allowing deduction of type and rank for the remainder of the items.

Wai-Mee Ching's APL compiler [Chi86, CNS89] requires the user to declare the arguments to the main function being compiled. YAT also permits declarations [GCDO86, GCDO87]. Even though declarations go against the grain of traditional APL design, their utility in obtaining good performance should now be obvious.

6.1 The Compilation Unit

APL compiler writers have taken two approaches to specification of the unit of code to be compiled. The separate compilation approach, used by Wiedmann [Wie83] and in ACORN [BBJM90, Ber90], permits compilation of single functions, such as those that are performance bottlenecks. It also permits convenient recompilation of small pieces of code during development. The ensemble approach, taken by Budd [Bud88], Ching [Chi86, CNS89], and Driscoll [GCDO86, GCDO87]. accepts an entire APL application as the compilation unit.

The single compilation approach offers speed and convenience, whereas the ensemble approach has the virtue of allowing interprocedural analysis to distribute morphological information about user-defined function parameters among the functions being compiled. This reduces the need for explicit declarations within the compilation unit.

The unique advantages of both approaches suggests combining them in a development platform environment capable of preserving interprocedural morphological information. This would provide the benefit of global morphological analysis, as well as permitting the compilation of single functions.

7 Identities

Well-written APL interpreters make substantial use of identities, simply because the cost of doing so is usually low and the potential gains are large. For example, catenation of an empty list to another list need not make a copy of the non-empty list as a result. Catenate merely recognizes the identity and passes the non-empty list as the result. Multiplication of an array by the scalar `1` can skip all mathematical calculations for the cost of one comparison.

Although it is possible to detect certain of these identities at compile time, it is not likely that many of them will be detected in practice, simply because people do not tend to write programs that way. Run-time detection is probably still the mainstay of identities in APL.

Partial properties, discussed in Section 14, offer the opportunity to reduce the cost of run-time identity detection and increase general system performance. In the case where a run-time check might detect an identity, it is possible that the check can be eliminated totally, or made simpler, because of information available from array annotation. In the case where a run-time identity clearly cannot occur, a run-time check for it can be eliminated.

8 Static Detection of Errors

Another potential benefit of array morphology is static detection of programming errors. For example, if an array is known to be rank `3`, and it is indexed with a rank `2` indexing expression, then the program is faulty. Similarly, a character array as an argument to logarithm is a bug in the source program.[1]

Annotators can make static conformability checks between verb arguments. For example, rotate can ensure that its left argument, if known, is integer-valued and of appropriate rank for conformability with the right argument. This technique has merit because any such check that can be performed at compile time need not have run-time code generated for it.

Other errors can be statically detected using knowledge of array properties. The potential benefits of such error detection are substantial and merit more study for both compilers and interpreters.

9 Performance and Triggers

Each iteration through the family handlers examines all entries, even though there may not be anything to gain by examining the majority of them. Annotation of `ls` takes 21 iterations to stabilize, so poor performance is highly visible. The treatment of all entries on each iteration was done for simplicity, since annotator performance is not a pressing issue for a research tool.

[1]Or, it is an ad hoc assertion's way of failing.

The question remains of how a production quality annotator can perform efficiently without having all family handlers know about the needs of each of the other handlers. One solution is to keep a *trigger* bit list marking each changed row, let the family handlers be dispatched based on need, and use the triggers to locate starting points for further annotation, thereby reducing the workload.

10 Assertions

Declarations are non-executable directives to a compiler, asserting truths about the program and its data. Although they work well for what they do, they are inadequate for more general assertions about a computation. *Assertions* are executable tests placed within an application to perform specific tests upon data, yet which have no effect upon the outcome of the application. When an assertion fails, however, it signals an event, to allow corrective action to be taken, or notifies the user or application writer of a problem.

Assertions are often used within applications to prevent or limit damage to databases by verifying certain properties of arguments before they are able to affect the database incorrectly. For example, a text database maintenance function might assert that its argument is textual, rather than numeric, data. A binary search with an argument that is defined as being sorted might issue an assertion that the argument is indeed sorted.

Assertions are also used to improve the locality of fault isolation. A large application may make assertions about its input, or about intermediate results, in order to halt execution immediately if the assertion fails. The other choice, of allowing execution to proceed, muddies the tracks of the original problem and makes fault determination more difficult.

Assertions and declarations have much in common. Assertions may be viewed as nothing more than executable declarations within a program. A program may assert that its argument is a table of floating point numbers by a sentence such as `assert (2=⍴⍴x)∧'f'=type x`. This is tantamount to a declaration in a compiled environment.

This suggests combining declarations and assertions into a single mechanism, denoted merely as *assertions*. The benefits of combining declarations and assertions into a single mechanism include:

- unification of two previously disparate concepts, simplifying the job of teaching and learning a language,
- providing a mechanism in which the work done by declarations can be done by assertions at either execution time or during the process of compilation,
- provision of a formal mechanism for providing declarations within APL, and
- specifying a formal mechanism for providing assertions within APL, as opposed to the ad hoc mechanisms now used by application writers.[2]

10.1 The Domain Conjunction

Discussions with Ken Iverson about assertions led to the concept of a *domain conjunction*, denoted here as `when`. Consider two verbs `work` and `valid`. The conjunctive expression `x work when valid y` returns `x work y` if the result of `x valid y` is `1`. Otherwise, it signals an event.

In compiled code, some of the work done by the domain conjunction could potentially be done at compile time, rather than being deferred until execution. For example, in the above-mentioned example of a binary search, it may be that the compiler is able to statically determine that the argument which is supposed to be in sorted order arose from an expression which guarantees its ordering. The code to make this validation at execution could then be elided. Contrariwise, if the argument was known to *not* be sorted, an exception could be raised at either compile or execution time.

Since assertion conjunctions are the same as any other conjunctions in the language, they can be added to existing dialects of APL and J with no changes in language syntax or semantics.

10.2 Assertions and Compiled APL

Assertions need not generate code to be effective. For example, assertions appearing as simple identities can have a substantial impact on potential code quality.

Consider the insertion of the following two statements into the `ls` function:

```
a←a[;]÷1.0
b←b[;]÷1.0
```

These are equivalent, in a morphological sense, to the floating table assertion made in Section 10. Annotation will detect the index as an identity, producing the entire array `a` as its result, with the benefit of implying that the array is of rank `2`. The divide by `1.0` is also detected as an identity, but has the effect of coercing the array type to floating.

In terms of array morphology, the effect of these statements is significant. As noted in Section 6, the effect of inserting declarations for the two function arguments dramatically increases the amount of morphological information obtainable.

[2]Currently, programmers write assertions using such diverse mechanisms as intentional divide by zero, branch to fractional line numbers, intentional syntax error, or vendor-dependent event signalling capabilities.

11 Constant Propagation

Constant propagation has not been taken very far in this project, partly because there was not much opportunity for it to occur in the small suite of programs used as test cases for the annotator. It will be interesting to perform array morphology on a larger collection of real world applications and see if this lack of opportunity is typical of APL programs in general, or if the test suite used thus far is anomalous.

12 Array Morphology and J

J is a generalization and rationalization of the ideas of APL [HIMW90, Ive96]. Many of the rough edges and anomalies of APL are gone from J, and it permits creation of purely functional programs. One of the anomalies that is gone is bracket indexing, which means that array annotation based on examination of indexing expressions can no longer make deductions of rank. J also removes most other rank restrictions on primitives. For example, indexof in J permits the left argument to be of any rank, rather than limiting it to rank `1` as APL does.

Do these generalizations and rationalizations, as desirable as they may be from the standpoint of language design, teaching, and convenience, present roadblocks to the annotation of J programs for compilation? Not if one accepts the need for assertions or declarations. In fact, many of J's features, such as static scoping, gerunds, and control structures, actually enhance compilability.

13 Roads Not Taken

No effort was made to examine the possibility of common subexpression elimination (CSE) [ASU86]. Common subexpressions appear most frequently in APL – as in Fortran – in subscript computations. Although common subexpressions may not occur as often in APL as in Fortran, their elimination may have even more value in APL than in other languages – APL's common subexpressions tend to be array-valued, rather than scalar-valued, so the potential for performance and storage gains is substantial.

Constant propagation and partial evaluation can be taken considerably further than they have been here. The calculus of j-vectors and addressing polynomials offer a large potential performance benefit.

Type-dependent optimizations are not presently supported. These optimizations include detected identities – the floor of a Boolean array need not generate code – and strength reduction: multiply on Boolean arguments can produce a Boolean result using logical and.

Another important area that remains totally unexplored is the morphology of recursive data structures, particularly in relation to adverbs and conjunctions. Support for such structures, known as boxed or nested arrays, is an area where the performance of current APL interpreters is generally deficient, having little support for any but the simplest of special cases.

14 Partial Properties

One of the by-products of array morphology is partial information about the properties of arrays. Although this area remains unexplored, it is potentially rich. The following are typical of the sorts of information that array annotation can provide:

- Indexing may determine some, but not all, of the elements of a shape vector.

- A verb may deduce that one of its arguments is numeric, but of unknown type.

- The index generator may determine that its argument is a singleton, but cannot determine its rank.

- An array may be known to be sorted in some order, a fact that could be exploited by search primitives.

- An array may be known to be a permutation of a dense set of integers.

- An array may be known to be a set of valid indices for another array.

Many other properties can be partially deduced. This is another unexplored area which merits additional research.

15 Summary

Array morphology is a powerful tool for the static determination of array properties in applicative languages including, but not limited to, type, rank, shape, and value. This information is invaluable in the creation of efficient compiled APL code.

When array morphology is inadequate, introduction of assertions or array declarations specifying critical array properties can be of great utility. The utility of array assertions suggests that a larger compilation unit increases the potential performance of compiled code, with no additional user-supplied information. Existing dialects of APL and J can support assertions as a domain conjunction, with no changes to the syntax or semantics of the languages.

16 Acknowledgments

This work was originally undertaken as a part of Professor Paul Chow's course, *Optimizing Techniques for Compiler-Generated Code*, at the University of Toronto's Department of Electrical and Computer Engineering. Professor Corinna Lee persuaded me to preserve the definition of basic blocks as they exist in other languages. Elena Anzalone provided valuable editorial services. Reviewer C09's comments improved the presentation.

References

[ASU86] Alfred V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[BBJM90] Robert Bernecky, Charles Brenner, Stephen B. Jaffe, and George P. Moeckel. ACORN: APL to C on real numbers. *ACM SIGAPL Quote Quad*, 20(4):40–49, July 1990.

[Ber87] Robert Bernecky. An introduction to function rank. *ACM SIGAPL Quote Quad*, 18(2):39–43, December 1987.

[Ber90] Robert Bernecky. Compiling APL. In Lenore M.R. Mullin, Michael Jenkins, Gaétan Hains, Robert Bernecky, and Guang Gao, editors, *Arrays, Functional Languages, and Parallel Systems*. Kluwer Academic Publishers, 1990.

[Ber93] Robert Bernecky. Array morphology. *ACM SIGAPL Quote Quad*, 24(1):6–16, August 1993.

[Bud85] Timothy A. Budd. Dataflow analysis in APL. *ACM SIGAPL Quote Quad*, 15(4):22–28, May 1985.

[Bud88] Timothy Budd. *An APL Compiler*. Springer-Verlag, 1988.

[Chi86] Wai-Mee Ching. Program analysis and code generation in an APL/370 compiler. *IBM Journal of Research and Development*, 30:594 – 602, 1986.

[CNS89] Wai-Mee Ching, Rick Nelson, and Nungjane Shi. An empirical study of the performance of the APL370 compiler. *ACM SIGAPL Quote Quad*, 19(4):87–93, August 1989.

[GCDO86] Jr. Graham C. Driscoll and D.L. Orth. Compiling APL: The Yorktown APL translator. *IBM Journal of Research and Development*, 30(6):583 – 593, 1986.

[GCDO87] Jr. Graham C. Driscoll and Donald L. Orth. APL compilation: Where does the time come from? *ACM SIGAPL Quote Quad*, 17(4):457–459, May 1987.

[HIMW90] Roger K.W. Hui, Kenneth E. Iverson, E.E. McDonnell, and Arthur T. Whitney. APL\? *ACM SIGAPL Quote Quad*, 20(4):192–200, August 1990.

[Int84] International Standards Organization. *International Standard for Programming Language APL*, ISO N8485 edition, 1984.

[Ive73] Eric B. Iverson. APL/4004 implementation. In *APL Congress 73*, pages 231–236. North-Holland Publishing Company, 1973.

[Ive96] Kenneth E. Iverson. *J Introduction and Dictionary*, J release 3 edition, 1996.

Abstract Syntax Tree				Annotations of `benchloop`							
				without declarations				with declarations			
Target	α	Verb	ω	Type	ρρ	ρ	Value	Type	ρρ	ρ	Value
`C0`				i	0		0	i	0		0
`C1`				i	0		2500	i	0		2500
`C2`				i	0		1	i	0		1
`L`				i	0		4	i	0		4
`n`											
`T_0`		⌊	5	i				i			
`T_1`		⍳	0	i	1	0		i	1	0	
`T_2`	7	⍴	6	i	0			i	0		
`r`		←	8	i	0			i	0		
`T_4`	1	×	5					i	0		
`n`		←	10					i	0		
`l:`											
`T_7`	11	−	2					i	0		
`n`		←	13					i	0		
`T_9`	0	<	14	b				b	0		
`T_10`	15	⍴	3	i	0			i	0		
`T_11`		→	16								
`r`		←	2	i	0		1	i	0		1

Figure 5: AST and annotations of benchloop. The AST axis and index have been elided, as has the annotation class.

[Jen70] M. A. Jenkins. The solution of linear least squares problems in APL. Technical Report Technical Report No. 320-2998, IBM New York Scientific Center, IBM Corporation, June 1970.

[SS88] Stephen Skedzielewski and Rea J. Simpson. A simple method to remove reference counting in applicative programs. Technical Report UCRL-100156, Lawrence Livermore National Laboratory, 1988. This report was prepared for submittal to the ACM SIGPLAN 89 Symposium on Programming Language Design and Implementation.

[Wie83] Clark Wiedmann. A performance comparison between an APL interpreter and compiler. *ACM SIGAPL Quote Quad*, 13(3), March 1983.