An Overview of the APEX Compiler

University of Toronto
Department of Computer Science
Technical Report 305/97
Robert Bernecky

Contents

1 Introduction **4**

2 Compiler Data Structures **7**
2.1 Compilation Unit Input 7
2.2 Compilation State Descriptor 8

3 Compiler Structure and Operation **11**
3.1 Tokenization and syntax analysis 11
3.2 Scope Analysis and Static Single Assignment Transformation 12
3.2.1 Calling Tree Construction and Depth-First Ordering 12
3.2.2 Semi-Global Analysis 12
3.2.3 Data Flow Graphs 13
3.2.4 Transformation into SSA Form 14
3.3 Data Flow Analysis 15
3.3.1 Array Predicates 16
3.4 Code Generation 17
3.4.1 Defined Function Generation 17
3.4.2 Primitive and Derived Function Generation 18
3.4.3 Macro Expansion of Code Fragments Using cpp 19
3.4.4 Macro Expansion of Code Fragments in APL 21
3.4.5 Macros and Special Cases 22

4 Parallelism within the APEX Compiler **25**
4.1 Tokenization 26
4.2 Syntax Analysis 26
4.3 Static Single Assignment Transformation 26
4.4 Data Flow Analysis 27
4.5 Code Generation 27

5 The APEX Dialect of APL **29**
5.1 Fundamental Restrictions 29
5.2 Design Restrictions 30

5.3 Implementation Restrictions 31
5.4 ISO Standard Compliance 32

Chapter 1

Introduction

APEX is an APL compiler that translates an extended subset of ISO Standard APL N8485 into SISAL, a functional vector language [Ber97]. APEX is written in ISO Standard APL, using nested array extensions and APL+Win flow control structures. APEX generates SISAL code which is then compiled into C by the Optimizing SISAL Compiler (OSC), and thence into machine code by the target system's C compiler. Figure 1.1 presents an overview of this division of labor. As shown in Figure 1.2, APEX is a multi-phase compiler, taking APL programs through the stages of tokenization and syntax analysis, semi-global analysis, static single assignment translation, data flow analysis, and code generation.

The APEX compiler comprises four major phases: tokenization and syntax analysis, static single assignment transformation and semi-global analysis, data flow analysis, and code generation. In order to gain an understanding of how APEX works, we will examine these four phases in turn. This will be facilitated by an overview of the major data structures used within APEX. That overview will be followed by a description of the compiler's phase-by-phase structure, and a discussion of parallelism within the APEX compiler itself. We close with a description of the dialect of APL implemented by APEX.

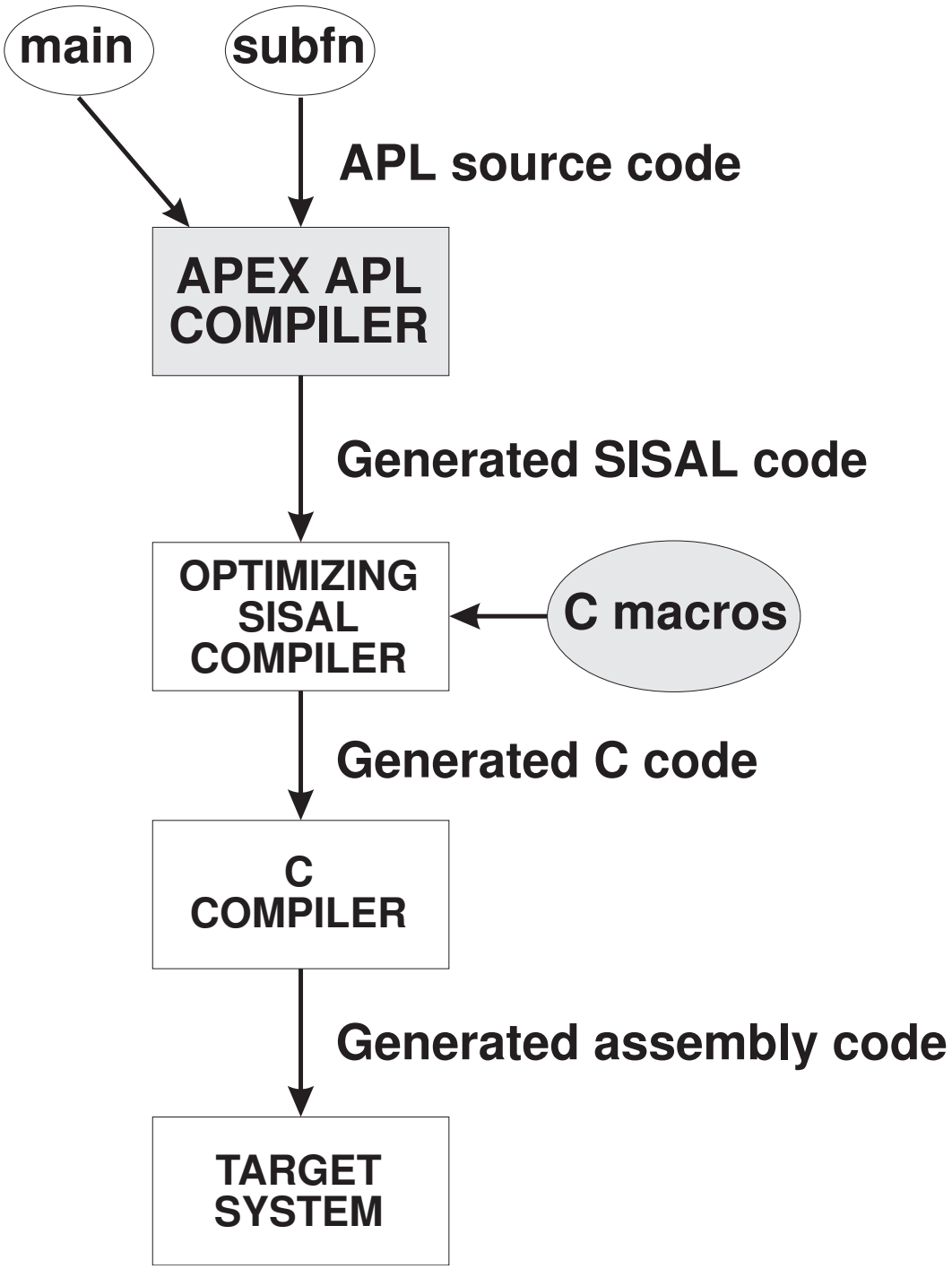

Figure 1.1: The Big Picture

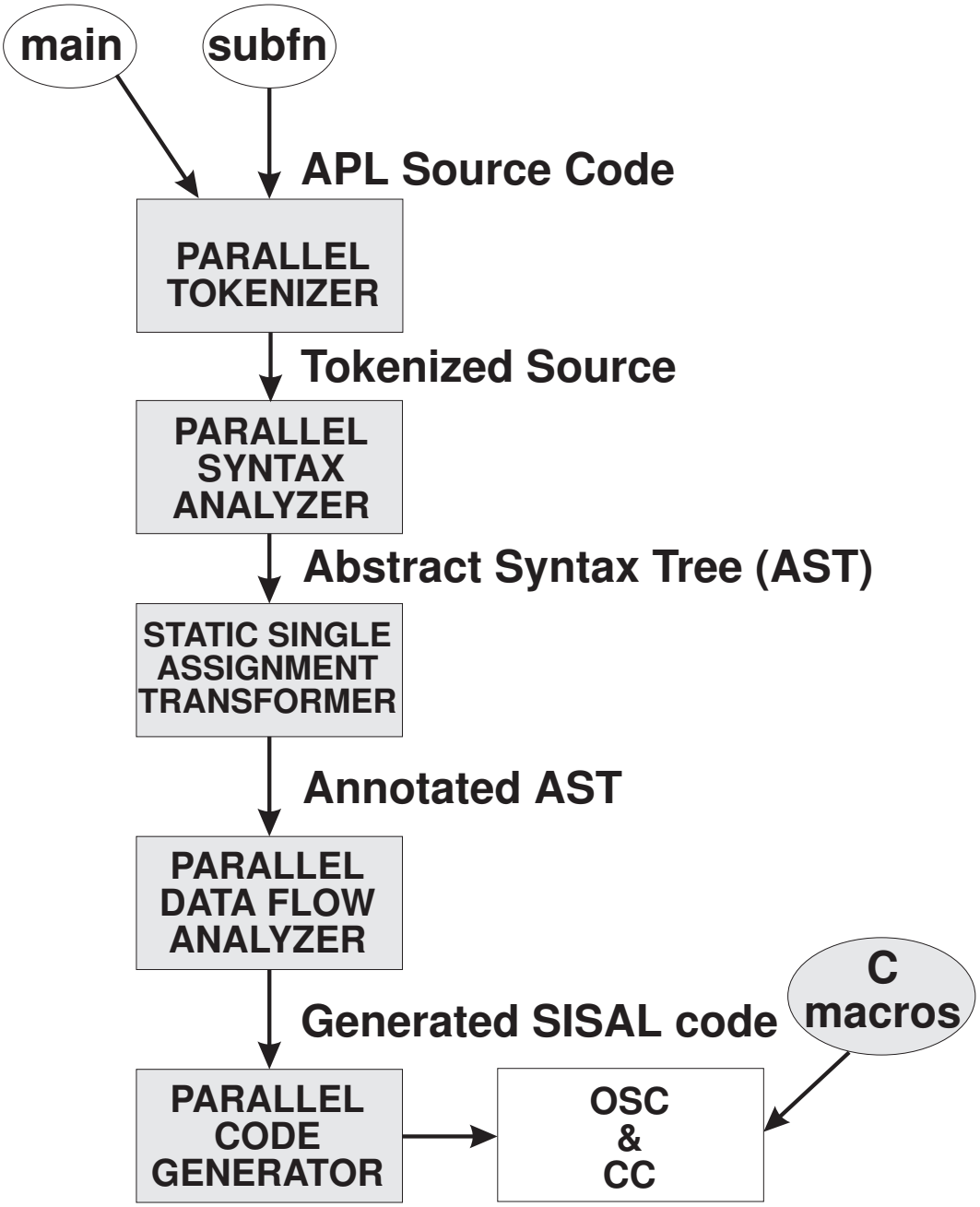

Figure 1.2: Structure of APEX

Chapter 2

Compiler Data Structures

Since a compiler is effectively a transformer of data structures, an understanding of its operation will be facilitated by a summary of the salient data structures and files associated with APEX. We will start with an examination of the input file, proceed through intermediate representations, and conclude with the compiler's output file. The data structures used within APEX are numerous, but most of them, being ephemeral, are not germane to a general understanding of APEX structure and operation. A few data structures, however, persist in APEX and serve as the mechanism by which information is passed from phase to phase within the compiler. We now introduce these files and inter-phase data structures.

2.1 Compilation Unit Input

The input to the APEX compiler is, of course, an APL program, comprising a set of defined APL functions. The compiler supports two methods of specifying this set. The first, direct, method is to present the compiler with a list of file names, with each file containing a single defined APL function. The second, indirect, method is to provide the name of a file that, in turn, contains a list of file names to be used as the compilation unit. The latter method is more suitable for compiling larger applications with numerous functions. [1] The representation of the text for a given function, obvious for ASCII-based languages, is not so clear for APL, so we briefly turn to that topic.

The APL community is cursed by the APL character set. The APL language requires a substantially larger character set than that provided by ASCII, but the vendors of APL have not standardized the mapping of glyphs to code points within that set. [2] The situation is somewhat like that existing

[1] It could also serve as a basis for an APL *makefile* maintenance facility.

[2] There is some hope that Unicode will provide a way out of this impasse, but the intransigence arising from extant application systems may make even that step a difficult one.

between ASCII-based and EBCDIC-based applications, except that the dichotomy here lies among language processors rather than hardware. A portable APL compiler, such as APEX, must have some way to wrest itself from this quagmire.

We took the approach of requiring all input to the compiler to be ASCII text, using Weigang's APLASCII encoding scheme [Dic95]. The APLASCII transliteration scheme converts an APL function from a host-dependent character set into a keyword-based ASCII representation of the function. Use of APLASCII establishes a firewall between APEX and the character set anarchy of current APL implementations. It does not, we point out, resolve all problems of character set dependencies *within* an APL application, but this has not been a problem with any of our text-oriented benchmarks.

In contrast to our file-based approach, some other APL compilers require the user to copy the compiler into the workspace to be compiled, or vice versa [CNS89, JO86, BBJM90]. Either of these approaches present problems that do not exist with the file-based approach. The first problem is that of name conflict. If a function name used within the compiler is identical to one used in the application, the act of copying will overwrite either the application function or part of the compiler, resulting in compilation of the wrong function or a compiler failure. The second problem is that of portability. The copy-based approaches require that the application workspace and the compiler both be present on the same APL system and same hardware. With APEX, we eliminate this restriction, permitting general cross-compilation.

2.2 Compilation State Descriptor

The *compilation state descriptor* (CSD) is the primary data structure used within APEX. The CSD is a list of arrays describing a single APL function. A typical application being compiled will be represented as a list of CSDs, one per defined function. The number of arrays in the CSD has grown as APEX has evolved. It presently consists of the following arrays: symbol table (ssast), abstract syntax tree (AST or ssaast), compression vector (ssacv), basic block partition vector (ssabb), control flow graph (ssacfg), dominator graph (ssadom), dominance frontier graph (ssadf), dominator tree graph (ssadomt), source text matrix (ssasrc), and tokenization matrix (ssatok). We will now discuss each of these elements in turn.

The *symbol table* is an N-by-3 nested array defining the names referenced in the function. The columns represent the name, syntax class, and scope of name, respectively. Most, but not all, of the function of the symbol table has now been merged into the AST.

The *abstract syntax tree*, or AST, is the most important single data structure used within APEX, as it contains the information about what the function actually does. The AST is built by the syntax

analyzer, amended and modified by the SSA and DFA phases, and used by the code generator.

The AST is, currently, an N-by-23 array, with at least one row per function and identifier that appear in the source function. A row of the AST, therefore, represents an identifier, a single APL primitive, or a derived or defined function call. Ancillary rows are used for purposes such as maintaining lists of semi-global parameters and lists of index expressions. Each column in the AST represents a specific morphological property. The AST is logically divided into two parts: *names* and *operations*. The first part contains entries for each defined function or variable referenced by the function. The second part contains operations that def or use named variables or temporary values. Like the CSD, the number of AST columns has increased with time. The columns currently present have the following definitions, with unknown or irrelevant entries containing the value *NULL*.

asttarget In the *name* section of the AST, *asttarget* is the actual name of the variable or function corresponding to this AST row. In the *operation* section of the AST, *asttarget* is the row index of the AST corresponding to the result of this operation. If this operation assigns a value to a name, *asttarget* will be the row index of some variable name in the *name* section of the AST. If this operation creates a temp, *asttarget* will be the row index of the operation itself.

astlarg If this operation is dyadic, *astlarg* is either the row index of the operation that generates its left argument or the row index of the name of its left argument.

astlop If this operation is a conjunction or adverb (operator), *astlop* contains the row index of its left operand (if the operand is a derived or defined function) or the function itself (if the operand is primitive).

astfn If this operation is a primitive, *astfn* is the primitive. Otherwise, it contains the row index of the defined function being invoked. Certain internally generated names within APEX also appear here; they are recognizable as single words prefixed with a colon.

astrop If this operation is a conjunction, *astrop* contains the right operand in the same form as *astlop*.

astrarg The right argument to this operation, in the same form as *astlarg*.

asttype The type of the result of this operation, expressed as an index into the `Types` vector.

astrank The rank of the result of this operation.

astshape The shape of the result of this operation.

astvalue The value of the result of this operation.

astclass The syntax class of this operation.

aststmt The line number within the source function matrix that generated this operation.

asttoken The column number within the source function matrix that generated this operation.

astxrho The element count of the array created by this operation.

asttag A utility cell used by various functions to identify AST rows so that morphological information can be extracted after operations that change the number of rows in the AST.

astlt The type of the left argument.

astlr The rank of the left argument.

astrt The type of the right argument.

astrr The rank of the right argument.

astparms A linked-list head, used to tie semi-global function parameters to the points of function invocation.

astscope The scope of a variable name in the *name* section: local, semi-global in, semi-global out, or semi-global in/out.

astpred A list of Boolean array predicates associated with the result of this operation.

The *compression vector* is a utility Boolean vector with as many elements as there are AST rows. It is used as a worklist in many operations involving the AST.

The *basic block partition vector* is a Boolean vector with as many elements as there are AST rows, with ones corresponding to AST rows that begin a new basic block.

The *control flow graph* (CFG) is a Boolean connectivity matrix with as many rows and columns as there are basic blocks in the function. The n^{th} row or column corresponds to the n^{th} basic block in the source function. Each 1 in the Control Flow Graph marks a possible path of control flow between the corresponding basic blocks of the function.

The *dominator graph* is identical in structure to the Control Flow Graph. Its entries indicate which basic blocks dominate others.

The *dominance frontier graph* is identical in structure to the Dominator Graph. Its entries mark the basic blocks corresponding to the dominance frontier of the source function.

The *dominator tree graph* is identical in structure to the Dominator Graph. Its entries mark the basic blocks in the dominator tree of the source function.

The *source text matrix* is the source program text, translated into APL glyphs and represented as a character matrix.

The *tokenization matrix* is a character matrix of the same shape as the Source Text Matrix, identifying the token class of each element of the source program.

Chapter 3

Compiler Structure and Operation

In order to simplify development and maintenance of APEX, its major phases are realized as four APL workspaces, chained together by `⎕load` functions. Phase-to-phase communication is handled by a workfile for each compilation unit. The duties of each of these phases – tokenization and syntax analysis, static single assignment transformation and semi-global analysis, data flow analysis, and code generation – will be discussed next.

3.1 Tokenization and syntax analysis

The tokenizer and syntax analyzer transform each user-defined function in the compilation unit into a CSD entry. The primary function of tokenization and syntax analysis is to create the AST, populated sufficiently to specify the complete semantics of the source function. The AST is, in a sense, a canonical representation of the function, as all comments and white space have been removed, yet the AST could theoretically be mechanically executed to produce the same result as the original source function. Since each function is compiled independently from the others, the only knowledge possessed by the syntax analyzer about other functions in the compilation unit are their name and valence (number of arguments).

Tokenization begins the compilation process. After the text of each source function has been read from file and converted from APLASCII form into APL glyphs, it is passed in matrix form to the tokenizer. The tokenizer analyzes the source text to produce a character matrix of the same shape as the source text, in which each character contains the token type of the corresponding source text character.

After tokenization, syntax analysis proceeds on a defined function by defined function basis. After

some preliminary analysis to extract localization information, the syntax analyzer analyzes each function line in parallel. Syntax analysis of each line is performed using a reduction parser that operates from right to left across the line. The parser utilizes a stack for holding state information and partial expressions, including intermediate values, stacked syntax states, and token classes. The output of this analysis is a set of rows to be appended to the AST. When all lines have been analyzed, their AST rows are catenated to form the AST for the entire function. Constants and declarations are handled at this point, amending the AST as appropriate. The symbol table, completed AST, source text, and tokenization information are appended to a workfile for subsequent use by later phases of compilation.

3.2 Scope Analysis and Static Single Assignment Transformation

The static single assignment (SSA) phase of APEX actually performs several duties, of which SSA transformation is the most significant. Other tasks performed by this phase include detection of semi-global variables and determination of their lexical scope, construction of the compilation unit calling tree, depth-first ordering of the compilation unit, and creation of data flow graphs. We now present these topics in the order of their execution by the compiler.

3.2.1 Calling Tree Construction and Depth-First Ordering

The first task undertaken by this compiler phase is construction of the defined function calling tree and transformation of the compilation unit into depth-first order. The former is a necessary prerequisite to SSA transformation; the latter is due to the code generation requirements of SISAL. As both of these are rather straightforward tasks, we will not dwell on them other than to mention that `BuildCallingTree` is non-iterative, but `DFSortast` iterates from the leaves to the root of the calling tree.

3.2.2 Semi-Global Analysis

The next task of the SSA phase is to handle semi-global variables within the compiler. Because SISAL is a purely functional language, a name cannot be passed from a function to a sub-function by means of lexical scoping or similar mechanisms. Instead, the passing of semi-globals in and out of sub-functions must be explicit. Several steps are required to do this properly. First, we identify defs and uses of semi-globals within each function and amend that function's AST to ensure that semi-globals are treated properly during data flow analysis and code generation. Second, we percolate the existence and status of each such semi-global upward in the calling tree to its point of localization. At

each level in the calling tree between these two points, the relevant AST entry is amended to make the semi-global an explicit parameter and explicit result of the intervening function(s). This action permits interprocedural data flow analysis to correctly propagate semi-global array morphology through the calling tree. To see why this is required, consider the following three functions:

```
r←main x;sg
 sg←⍳x
 r←subfn x
 r←r+sg

r←subfn x
 r←leaf x

r←leaf x
 r←⌽x
 sg←r+1
```

Interprocedural data flow analysis propagates array morphology from caller to callee and vice versa. Since `subfn` does not def or use `sg`, there is no way for the morphology of `subfn` to be passed from `main` to `leaf` and back unless `subfn`'s symbol table is amended to include the information that `sg` passes through its invocation.

Once the semi-global marking is complete, interprocedural data flow analysis propagates the morphology. For a function call, the morphology of the caller's arguments are propagated down into the called function. The caller's SemiGlobalIn and SemiGlobalInOut informal arguments to the callee are treated in the same fashion. The morphology of the callee's explicit result is propagated up to its caller. The callee's SemiGlobalOut and SemiGlobalInOut informal results are treated similarly.

At the end of this stage, the APL application being compiled has been transformed into a state in which *all* interprocedural references are explicit, rather than implicit. APEX displays informative messages about such implicit variable use during this stage of compilation. Since a common source of bugs in APL applications is faulty localization of variables, such diagnostic messages may be of utility to programmers, even if they have no intention of actually executing the compiled code.

3.2.3 Data Flow Graphs

The next stage of this compiler phase generates the control flow and data flow graphs and vectors required by the SSA transformer. The Basic Block Partition Vector is built first, followed by the Control Flow Graph and the various dominator graphs. We represent these graphs as N-by-N Boolean matrices; APL functions are small enough – often consisting only of a single basic block – that the quadratic costs implied by this representation are inconsequential. The `MarkBasicBlocks` and

`GenerateCFG` functions are non-iterative, executing quickly in interpreted APL. The `Dominator` function iterates over the control flow graph to the depth of the calling tree.

3.2.4 Transformation into SSA Form

The major task performed by the SSA phase of compilation is the transformation of APL functions into static single assignment form, in which each variable name appears but once as a target of assignment. This is done for several reasons: First, it simplifies data flow analysis considerably. Second, it is a requirement of the SISAL code generator. Third, SSA offers more precise data flow information than is available with def-use chaining. The method we follow for implementing SSA renaming is that described by Wolfe [Wol92, CFR*89]. We first reduce the amount of renaming to be done by removing from consideration all variables in the source function that contain only one instance of a def, since these are, by definition, single assignment.

The step after winnowing of single assignments is to introduce ϕ-functions at each confluence of control flow in the source function. This is done on a worklist basis, based on the dominance frontier. Since the dominance frontier demarcates the point where domination ceases, two or more defs of the same name may conjoin there. A ϕ-function may be thought of as a function that picks the correct def, depending on the flow of control that got us to the confluence. These ϕ-functions serve a notational and analysis purpose only. They do not appear in the generated code, although they are crucial to generation of SISAL loop structures from iterative APL functions.

After ϕ-functions have been placed, all variables containing multiple defs are renamed appropriately, to bring the source function entirely into SSA form. In the interest of expediency, our implementation renamed one variable at a time. Execution time for this phase of compilation would benefit from rewriting the SSA transformer to handle all variables in parallel.

In the interest of improving the performance of this phase, we made a minor change to the renaming algorithm. We mark all defs in parallel as a pre-processing step, then perform all renames of uses in parallel within each basic block. The algorithm described by Wolfe assumes a line-by-line traversal of each basic block. Since the remainder of the algorithm is fairly faithful to that of Wolfe, we shall not describe it further.

We encountered one problem with SSA and iterative APL that we were unable to resolve in a way that we consider satisfactory. Consider an iterative APL function such as the following:

```
R←InitialValue N
:for i :in ⍳N
R←i
:endfor
```

This function performs perfectly as long as `N` is non-zero, but will terminate with `value error` if `N` is zero, because `R` will never have been assigned a value. SISAL's semantics, by comparison, make it impossible for a function to complete execution without each of its results having a value. Hence, `value error` cannot occur in SISAL. SISAL achieves this by requiring that all loops include an assignment to `R` of an initial value as part of the loop header, even if the loop is guaranteed to always execute at least one iteration. In the event that the loop terminates immediately – a zero iteration loop – that initial value becomes a formal result of the loop construct. Thus, there is a semantic mismatch between APL and SISAL in this regard. We circumvented this mismatch in the following way: if the relevant variable has an extant value upon entry to the loop, we use that value as the initial value in the loop header. Otherwise, we concoct an initial value of an array of fill elements of the same type and rank as the value def'd within the loop. This enables us to generate compilable SISAL code, but does introduce a potential problem of an erroneous result: execution of the compiled version of `InitialValue` with an argument of zero would complete normally, returning an incorrect result. We could introduce a flag bit into the initialization and loop structure of such loops to force an error in the event that the loop completes with no iterations, but have not done so thus far.

3.3 Data Flow Analysis

Another key component of APEX, from the standpoint of generated code performance, is its data flow analysis phase, for it is here that APEX deduces the morphological information needed for generation of efficient code. The primary task of the data flow analysis phase is to determine the rank and type of arrays created by *all* operations in the source function. If these facts are not known, the code generator cannot proceed.

The secondary task performed by data flow analysis is to determine ancillary morphological information. Such information is not strictly needed to perform code generation, but it may enable generation of higher-performance code. Ancillary morphological information currently includes array predicates, array shape, array element count, and array value.

Data flow analysis in APEX is a multi-pass operation that iterates until a fix point is reached. A semi-lattice associated with each deduced property ensures that such a fix point will eventually be reached.

Global data flow analysis is performed in parallel on each function in the compilation unit. At the end of one pass of data flow analysis on each function in the unit, interprocedural data flow analysis is performed serially, passing morphological information up and down in the calling tree. At the end of interprocedural DFA, a check is made to see if the morphology of the compilation unit has reached

a fix point. If so, data flow analysis can do no more work, and an assertion is executed to confirm that type and rank values are known for all arrays created in the execution of the compilation unit.

Global data flow analysis executes a series of functions that operate on the AST, controlled by a compression vector worklist. The method we use is that described by Saal and Weiss, Budd, Ching, and others [Bud88, Chi86, Ber93, WS81]. Basically, the function that performs data flow analysis for a particular class of primitives (*e.g.*, dyadic scalar functions) examines the rows of the AST marked by the worklist to see if their operations fall within its purview. For any such operations, it then checks to see if the ranks or types of their arguments are known. If they are known, then the DFA function uses class-specific rules to determine the result type and rank, then merges this information into the AST. The function then clears worklist entries corresponding to the entries it has examined and sets worklist entries that correspond to AST rows that reference the AST rows whose properties have just changed.

Interprocedural DFA is fairly straightforward. It extracts the morphology of interprocedural parameters and results, all of which are now explicit, thanks to SSA and semi-global analysis, and propagates that morphology down to called sub-functions and up to callers. Worklist elements are set appropriately for affected AST rows for each function, after which another round of DFA is performed.

Data flow analysis is currently the slowest phase of APEX. As DFA is highly iterative and operates on fairly small arrays, interpretive overhead dominates its execution time. The speed of type and rank determination could improved somewhat by adoption of a table-driven algorithm, where appropriate. This would reduce the number of APL primitives executed per iteration. Additional performance improvements could also be achieved by abandoning the nested array AST structure, at least within the DFA phase, and using flat arrays and marker arrays instead. However, as this change would likely obfuscate the code structure, we are reluctant to make such a change, and plan instead to bring APEX to the point where it can compile itself.

3.3.1 Array Predicates

Array predicates are discussed in our thesis [Ber97]. The predicates currently supported by APEX data flow analysis are shown in Figure 3.1. The rules for generation and propagation of predicates across function boundaries are presented in Figure 3.2.

Predicate	Description	Definition (□ct=0)
astPredPV	array is permutation vector	∧/p∈⍳⍴p
astPredPVSubset	array is subset of permutation vector (may contain duplicates)	∧/p∈⍳⌈p
astPredNoDups	array elements are unique	∧/(⍳⍴p)=p⍳p
astPredAll2	array is formed from all 2s	∧/p=2
astPredSortedUp	array is in non-descending order	p≡p[⍋p]
astPredSortedDown	array is in non-ascending order	p≡p[⍒p]
astPredKnowValue	array value is known at compile time	
astPredNonNeg	array elements are all non-negative	∧/,p≥0
astPredInteger	array elements are all integer-valued	∧/,p=⌊p

Figure 3.1: Definitions of array predicates

3.4 Code Generation

The sole duty of the APEX code generator is to generate the most efficient SISAL code possible for each compilation unit, based on information contained in the AST as annotated by the SSA and DFA phases. This phase of compilation is divided into two parts, one that emits the code associated with user-defined functions and one that emits definitions of APL primitive and derived functions that are common to the compilation unit. We will examine the generation of defined functions and their invocations of primitives first, then turn to generation of primitive and derived function definitions.

3.4.1 Defined Function Generation

The code generator emits code for each user-defined function in the compilation unit. APL constants are converted to SISAL format at this stage of processing, and the header and trailer boilerplate to be wrapped around the generated code is created. At this point in time, loop-carried value transformations are also performed.

Selective renaming of loop-carried variables within `:for` loops represents a major transformation performed by the code generator. This requirement arises because SISAL, being a single-assignment language, allows a programmer refer to a variable's value from the previous iteration by prefixing the variable name with the keyword "old". The code generator has to perform a substantial amount of work here. First, it has to establish an initial value for the loop-carried value entering the loop. It then has to adjust references to the previous iteration's value to be prefixed with the "old" keyword, but to leave unaltered any references made after the def of the loop-carried value. The ϕ-functions described earlier are the basis of this work. This code has to work outward through nested loops, and also has to generate code to explicitly export all newly def'd values from loops, as SISAL loops are purely

functional. This implementation of this transformation remains a troublesome and particularly fragile area of the code generator.

After loop-carried value transformation is complete, APEX produces the code that invokes APL primitive and defined functions within each user-defined function. This apparently straightforward task is complicated by a number of factors, including the presence of semi-global variables and of derived functions that have user-defined functions as operands. As an example of the latter, consider the situation if two user-defined functions execute the reduction expression `f/Y`, where `f` is a user-defined function. There is the possibility that the function `f` is local to each of those functions and, therefore, represents different functions. If this is the case, we must generate separate code for each reduction, reflecting their distinct operands. We chose to generate these reductions locally, restricting their name scope to that of the function invoking the reduction. This forces derived function code generation to be performed at two different points, one for those with primitive operands and one for those with defined operands. We are not satisfied with this approach, as it has resulted in a number of code faults arising out of confusion over this division of labor.

The above issues notwithstanding, most of the work of this part of the code generator is production of function calls and macro invocations for primitive functions. The APEX code generator emits one such call for each primitive, defined, or derived function call in the source function – approximately one per AST operation row. A typical call – they differ somewhat, depending on the class of the function being invoked – appears in Section 3.4.3. If the function being invoked has semi-global arguments or results, they are now appended to the invocation as appropriate.

3.4.2 Primitive and Derived Function Generation

The second major step in code generation is production of SISAL functions that implement specific cases of APL primitives, *e.g.*, *BooleanVector* + **.** × *IntegerMatrix* or ⍉*Double-RealMatrix*. This step follows the creation of calls to such functions performed by the first step of code generation. After the calls are generated, the compiler makes a list of the intersection of all primitive and derived function invocations in the compilation unit. This serves as the argument to the second step, which then generates a single copy of each function in that list. This substantially reduces the size of the SISAL code generated by APEX, but does not materially affect the size of the generated C code, as OSC tends to inline all function calls.

At present, the generation of SISAL code implementing APL primitives is handled partly by macro expansion and partly by generation of defined SISAL functions, depending on the class of primitive function being generated and the history of code generator evolution. For example, the dyadic scalar function generator emits inline code via macro expansion, but the older monadic scalar

function generator emits defined SISAL functions and calls to those functions. We will discuss these two approaches in turn.

3.4.3 Macro Expansion of Code Fragments Using cpp

The fundamental approach we took to code generation was inspired by macro-assemblers and several dynamic code generators we had written in the past for APL interpreters. The SISAL code for each primitive is written as a code fragment with appropriate parameters for text replacement. These fragments are assembled piecemeal to produce a functioning piece of SISAL code by use of the C preprocessor, cpp, or by an APL macro expander.

To get a feel for how the macro facility works, consider an APL expression to add two vectors `vi+vd`, one integer and one double-real. For simplicity, we assume that the two arrays are the same length, so that no conformability check is required. This expression would be invoked as follows, with required code fragments appearing next:

```
tmp72 := dsfctl111sl(dplus,D,D,vi,I,integer,vd,D,double-real)
```

Description	Fragment text
Double-real addition	`dplusDD(XV,YV) (XV+YV)`
Vector-Vector scalar function control	`dsfctl111sl(FN,CT,ZT,x1,XT,XTYPE,y1,YT,YTYPE) (for x0 in x1 dot y0 in y1 returns array of dsfctl000(FN,CT,ZT,x0,XT,XTYPE,y0,YT,YTYPE) end for)`
Scalar-Scalar control	`dsfctl000(FN,CT,ZT,x0,XT,XTYPE,y0,YT,YTYPE) FN##CT##ZT(XT##to##CT(x0),YT##to##CT(y0))`
double-real to double-real coercion	`DtoD(x) (x)`
integer to double-real coercion	`ItoD(x) double_real(x)`

The invocation name `dsfctl111sl` indicates a dyadic scalar function (`dsfctl`), argument and result ranks of 1 (`111`), and a special case marker (`sl`) indicating conformable (Same Length) arguments. The arguments to the invocation are the scalar function being executed (`dplus` for dyadic plus); the compute type and result type of the function (`D` for double-real); the left argument (`vi`) along with its type (`I` and `integer`) expressed in two forms because the macro expander is a pure text replacement facility with no computational capability; and the right argument (`vd`) along with its types (`D` and `double_real`). The `#` characters appearing in the macro definitions, an artifact of the cpp macro-expander, ensure that no white space appears in the generated names. The final generated code looks like this:

```
(for x0 in vi dot y0 in vd returns array of
  double_real(x0)+y0
end for)
```

In the above example, we made a trivial use of recursive macro invocation. We found that this substantially simplified the code generator and reduced the number of code fragments by a healthy margin. For example, without recursive code generation, we would need N^2 code fragments to handle just the integer-integer cases of scalar functions on arguments of rank less than N. With recursive generation, we need only N of them. We emphasize that this recursion is handled entirely within cpp and not in the APEX code generator. Thus, we use existing software tools to maximum advantage.

We started to look into recursive code generation as a result of problems with generation of code for indexed reference and indexed assign. At that time, we were attempting to generate complete code for each single primitive operation in isolation. This worked adequately, but required substantial amounts of hand-coded support for exceptions and special cases. When we began to generate code for APL indexing, we ran into trouble. The semantics of APL indexing are complicated in the extreme, allowing for an almost arbitrary number of arguments; the conformability rules for indexed assign are quite baroque. Attempting to create a correctly functioning, let alone efficient, code fragment to implement APL indexing is a highly complicated task. We eventually came to the realization that indexing can be recursively performed one axis at a time, starting at the leftmost axis and working towards the rightmost, reducing the rank and complexity of the indexing expression at each point. This led us to generate recursive indexing code which, although entirely coded in APL, was substantially simpler than the brute force approach we had taken earlier. Upon reflection, we realized that the recursive generation approach considerably simplified the entire code generation process. That realization, combined with an understanding of the OSC pre-processor, led to a fairly general recursive macro expansion methodology for code generation.

The macro expansion facility was originally written entirely in APL, with code fragments in ASCII text files written in a form suitable for convenient editing and automatic extraction. As we became familiar with the use of cpp as a front end to OSC, we converted parts of the code generator to use cpp. This both simplified and sped up code generation, but it made generated code much harder to read and exacerbated the difficulties associated with tracing of run-time faults. At the same time, we coincidentally replaced generated SISAL sub-function calls to APL primitives with inline code, because it happened to be a convenient way to implement the macros. We soon found that inlining at this point substantially increases the size of generated SISAL code and makes it very difficult to use code profilers to compare the performance of two run-time algorithms. [1] In spite of these problems,

[1] Since OSC normally inlines all functions, this does not affect the size of the generated C code.

we lean toward the macro generation scheme as our long-term direction of development.

One other problem associated with inline code generation is name conflicts. In our original scheme, we generated function calls for scalar function invocations. When we began to generate this code inline, we encountered the problems described in the previous paragraph, as well as an insidious problem introduced by the change in semantics from a formal function call with named parameters to text replacement with no renaming: subtle semantic errors can easily be introduced into the generated code. For example, if a function code fragment uses the variable `K` as a loop counter, and the argument to the function is also called `K`, then the reference generated within the SISAL code will be to the loop counter, rather than to the argument. This problem can probably be dealt with by adopting strict naming conventions for names used within and without macros. However, for this and the reasons cited earlier, we plan to modify our invocation and generation procedure to use explicit function calls, while maintaining the flexibility of cpp to generate the primitives themselves.

The major problem we have with use of cpp is that it is a purely mechanical text replacement facility. It is not possible to perform table-lookup operations or to choose one set of expansion text over another based on a parameter. This needlessly complicates the invocations of primitives and makes certain types of code generation difficult or impossible.

3.4.4 Macro Expansion of Code Fragments in APL

Those functions that are not macro-generated by cpp are generated using an APL-coded selection and replacement scheme that provides more sophisticated capabilities than cpp. In particular, the APL macro expander permits indexed selection of one code fragment from a set of related fragments. This permits selection of special case algorithms for particular cases of a primitive. For example, the *indexof* function currently permits generation of three different algorithms for character arguments: a generic case, a special case for `⎕av⍳characters`, and a special case for `characters⍳characters`. The code generator selects a case number based on an analysis of the source code; the macro expander selects the appropriate case for generation. The case selector is also used for inner product generation, to pick the *STAR*, *TransposeRight*, or *QuickStop* cases.

The APL macro expander also supports a pattern-matching selector that picks the code fragment that is, in some sense, *closest* to a goal. The pattern matcher was used in certain cases to automatically select a special case, based on argument types, in lieu of the general case. A pattern matcher that provides the closest match to a particular case has the virtue of permitting incremental addition of special case code fragments to a compiler fragment library without requiring concomitant changes to the compiler itself. For example, a generic *multiply* code fragment could be augmented, at a later date, by writing a special case for Boolean multiply that used *logical and*, then placing the special

case fragment before the generic fragment and limiting its range to Boolean arguments by making appropriate declarations in the code fragment header.

The two pieces of code generated above – function definitions and their invocations – are catenated together with appropriate header and trailer boilerplate, then written to a host system file for passing on to OSC for translation to C.

3.4.5 Macros and Special Cases

The adoption of macros and code fragments as the fundamental mechanism for code generation facilitates the generation of special case code. Such special case code can outperform generic code by a substantial margin, due to exploitation of morphological information obtained from data flow analysis. For example, in a dyadic scalar function, shape information may tell us that the two arguments are conformable; an array predicate may inform us that the argument to the *index generator* function is a non-negative integer. In the first case, code for length error checking can be removed; in the second, domain checking code can be removed.

In an agglomeration-based code generation scheme, each special case requires substantial code generator modifications to create the particular text of that case. A macro-based code generator, by contrast, need only identify the special case to the macro expander. We will use the *index generator* as an example of two methods of doing this.

The first method is to make two copies of the code fragment, with slightly different names:

Fragment name	Fragment text

Generic	`iotax01sy(y0,YT)`
	`(for i in QUADIO,QUADIO+`
	`(ConformNonNegativeInt(YT##toI(y0))-1)`
	`returns array of i end for)`

Non-negative scalar	`iotax01NonNegsy(y0,YT)`
	`(for i in QUADIO,QUADIO+`
	`(YT##toI(y0)-1)`
	`returns array of i end for)`

The first fragment, `iotax01sy`, operating on a generic scalar argument of unknown value, requires checking to ensure that the argument value is a non-negative integer. This is implemented by the `ConformNonNegativeInt` call. The second fragment, `iota01NonNegsy`, is identical to the first except for the elision of the domain error check. The second fragment is used when array predication or other morphological information has informed the code generator that the argument to *iota* is known to be a non-negative integer. To generate this second fragment, all that the code generator need do is append the suffix `NonNeg` to the code fragment name when appropriate. The `sy` suffix is

appended when another predicate has determined that the right argument is a scalar. Thus, the code generator merely builds names by appending suffixes to fragment names; the macro expander does the rest of the work. In the case where a specific special case fragment has not been written yet, an alias fragment, invoking the generic case, is all that is needed in the fragment library.

An alternate method of handling special cases involves addition of parameters to the invocation. In the above example, the special case might be handled by adding an error-checking function to the invocation:

```
iotax01sy(y0,YT,check)
(for i in QUADIO,QUADIO+
(check(YT##toI(y0))-1)
returns array of i end for)
```

The generic invocation would then be `iota01sy(N,I,ConformNonNegativeInt)`, while the special case invocation would be `iota01sy(N,I,'')`. The latter case would not perform any error checking.

Both of these approaches have their merits, depending on how different the special case code is from the generic code. They possess the common trait of easing the task of generating special-case code for a complex language.

APL operation	Predicate name PV	PV Subset	No Dups	All2	Sorted Up	Sorted Down	Know Value	Non Neg	Integer
x⊤y	0	0	0	0	0	0	0	0	0
x⊥y	0	0	0	0	0	0	0	0	0
x,y	0	0	0	and	0	0	0	and	and
x f⍤ k y	0	0	0	0	0	0	0	0	0
x f⍤ g y	0	0	0	0	0	0	0	0	0
x⍳y	0	0	0	0	0	0	0	1	1
x⍧y	0	por PV	p	p	p	p	p	p	p
x⊆y	0	0	0	0	0	0	0	1	1
x∈y	0	0	0	0	0	0	0	1	1
x\y	0	0	0	0	0	0	0	p	p
x/y	0	por PV	0	0	0	0	0	0	0
:for	0	por PV	p	p	p	p	0	p	p
x⍋y	1	0	1	0	0	0	0	1	1
x⍒y	1	0	1	0	0	0	0	1	1
x[i]←y	0	0	0	p	0	0	0	p	p
x[i]	0	por PV	0	p	0	0	0	p	p
⍳y	1	0	1	0	1	0	0	1	1
,y	p	p	p	p	p	p	0	p	p
f/y	0	0	0	p	0	0	0	0	0
f\y	0	0	0	p	0	0	0	0	0
x/y	0	0	0	p	p	p	0	p	p
xρy	0	0	0	p	0	0	0	p	p
x↑y	0	0	0	0	0	0	0	p	p
⌽y	p	p	p	p	0	0	0	p	p
x⌽y	0	por PV	p	p	0	0	0	1	1
x f.g y	0	0	0	0	0	0	0	0	0
x ∘.g y	0	0	0	0	0	0	0	0	0
ρy	0	0	0	0	0	0	0	1	1
ϕy	p	p	p	p	nsd	nsu	0	p	p
xϕy	p	p	p	p	0	0	0	p	p
+y	0	0	0	0	0	0	0	0	0
x+y	0	0	0	0	0	0	0	0	0

0	clears predicate
1	sets predicate
p	propagates predicate from appropriate argument
and	logical and of predicates for both arguments
por	logical or of argument predicate with named predicate
nsu	~SortedUp
nsd	~SortedDown

Figure 3.2: Array predicate generation, invalidation, and propagation

Chapter 4

Parallelism within the APEX Compiler

The use of APL as an implementation language for a compiler naturally encourages parallel expression of the compilation process. This property of APL enabled us to conveniently introduce a substantial amount of coarse- and fine-grain parallelism into APEX. In fact, writing parallel code in APL is usually easier, quicker, and more reliable than writing iterative or serial code.

The fundamental tools used by the compiler to achieve this end are the Extended ISO APL *each* adverb, to perform SPMD computation in a *forall* manner, and the array semantics of APL, to perform SIMD computations. Unfortunately, since we do not yet have the data flow analysis capability required to compile code for the nested array structures used by APEX, we cannot yet use the compiler to compile itself. Therefore, we are unable to quantify that parallel speedup and the actual level of parallelism obtained within APEX itself. Nonetheless, as we feel that our approach may be applicable to other compiler development efforts, we take this opportunity to present it.

The compilation unit for APEX is a list of defined APL functions. In many parts of the compiler, each user-defined function is manipulated in parallel, as was done by Wortman and Junkin [WJ92]. As noted above, APL encourages this kind of parallel expression, as programs are usually expressed in a more concise and more readable form if parallelism is used in lieu of iteration. In this form of SPMD parallelism, each function is treated independently; synchronization occurs only at the end of all processing of all functions. Of course, within this coarse-grain parallelism, APL's fine-grain, array-based parallelism is exploited as well.

We will now present a brief walk-through of some of the more salient parallel structures in APEX. A detailed analysis is precluded by the considerable amount of parallel expression within the compiler. The structure of our presentation will mirror the phase structure of APEX.

4.1 Tokenization

The tokenization phase of APEX operates at two levels of parallelism. It tokenizes each function in the compilation unit with SPMD parallelism, by use of the *each* operator (¨) of Extended APL (`tokenize¨cu`). In this context, the *each* operator may be thought of as an N-way fork, providing a parallel thread of execution for each function in the compilation unit. Within each of these resulting threads, a single function is tokenized using a single-pass (in the APL sense), SIMD tokenizer. The tokenizer itself is straight-line APL code containing no explicit loops. It makes extensive use of SIMD expressions including Boolean mask operations, scans, and partitioned scans and reductions [Ive62, Smi79].

4.2 Syntax Analysis

Like the tokenizer, the APEX syntax analyzer also exhibits parallelism at several levels. It operates in SPMD parallelism on each function in the compilation unit. Within each such thread of execution, syntax analysis is performed in SPMD parallelism on each line of each function. Within each function line, as one might expect, syntax analysis is done serially, a token at a time. Although this was the most convenient way to write the syntax analyzer, it also obviously provides an abundance of parallelism for this phase of compilation – one thread per function line within the compilation unit.

4.3 Static Single Assignment Transformation

All compiler phases after syntax analysis offer an added outer level of parallelism: Multiple compilation units presented to these phases are treated with SPMD parallelism. This was a trivial enhancement, added during compiler development as a convenience, to facilitate rerunning of specific compiler phases on multiple benchmarks. The remainder of this section will present compiler parallelism as if there were only one compilation unit: it is understood that this extra layer of coarse-grain parallelism exists in addition to those being discussed. The syntax analysis phase does not currently support the multiple compilation unit capability, but such support could be written in a few minutes.

Within the thread of execution for a given compilation unit, static single assignment transformations and related operations are performed in a partially serial and partially parallel fashion, as appropriate. Construction of the compilation unit calling tree is performed in two stages: a SIMD within SPMD stage extracts the names of defined functions called by each function. These are combined into a single calling tree in a subsequent serial stage. The compilation unit itself is sorted into

the depth-first ordering required by OSC by a serial step operating on the calling tree.

With two exceptions, all remaining stages of the SSA phase of compilation are performed with SIMD within SPMD parallelism. There is no explicit looping within any of the functions that compute basic blocks, control flow graph, dominators, dominance frontiers,

There is explicit looping in the following functions, because they contain dependencies on ordering across basic blocks, defined functions, or both: `SSARename`, `SemiGlobalAnalysis`, and `SemiGlobalUnref`. The amount of parallelism in these could be increased substantially with a bit of development work, coincidentally improving their interpreted performance. Specifically, `SSARename` currently renames one variable at a time, looping over all basic blocks in a function to do so. Revising that function to rename all variables at once would remove a compile-time bottleneck that becomes apparent when compiling very large functions, such as **metaphon**, that have many basic blocks and a large number of local and semi-global variables.

4.4 Data Flow Analysis

The data flow analysis phase of APEX operates in parallel on each compilation unit. Within a given compilation unit, data flow analysis works in a barrier-synchronized parallel fashion on each function in the unit. That is, global data flow analysis is performed using SIMD parallelism on each function in the compilation unit, then a serial step performs interprocedural DFA between the elements of the compilation unit. Within the global DFA stage, execution is serial by function type, with SIMD parallelism for each function type handled by DFA. For example, data flow analysis for all dyadic scalar functions within a user-defined function is performed in SIMD parallelism by the function `dfadsf`.

4.5 Code Generation

The APEX code generator, as noted previously, handles multiple compilation units in parallel. Within a given compilation unit, code generation is performed in two stages. The first stage is performed with SPMD parallelism across each defined function. This stage emits the SISAL code generated for defined functions. A second stage emits the SISAL code that defines those APL primitives not already generated as inline code. This stage is performed in serial SPMD fashion, for two reasons. First, it generates only one copy of each sub-cased APL primitive in the entire compilation unit, requiring generation of the set of all such primitives in the compilation unit. This, in turn, requires access to the output of the defined function generation stage, which may have introduced additional

primitive invocations. Second, although emission of these primitives could, in principle, be performed in parallel, APL does not provide any clean mechanism for performing MIMD parallelism. [1] Hence, we serialize this stage of the compiler.

The above summary of parallelism within APEX should make it apparent that an abundance of available parallelism exists within compilers, or at least within a compiler for APL. One of our tasks for the future is to get the compiler to the stage where it is able to compile itself, at which point we hope to quantify this claim.

[1] Proposals have been made for introducing MIMD capabilities into APL, but only the J language has actually implemented them as of this date [Ber84, BH91].

Chapter 5

The APEX Dialect of APL

Any implementation of a computer language, whether compiled or interpreted, embodies concrete design decisions made manifest by the dialect of the language that it supports. This section describes the APL dialect that APEX supports, in terms of known deviations from the ISO APL Language Standard N8485 and extensions beyond the Standard [Mor84]. Our presentation of restrictions and deviations will follow Ching's taxonomy of fundamental, design, and implementation restrictions [CNS89].

5.1 Fundamental Restrictions

Fundamental restrictions are those that are imposed on a language by the nature of compiled code. For example, creation of new functions during execution via ⎕fx would require a complete interpreter to be part of the run-time compiled environment. Our fundamental restrictions are similar to those adopted by the designers of other APL compilers [Bud88, CNS89, JO86, BBJM90].

Functions cannot be created dynamically within APEX-generated code, nor can identifiers change syntax class, except from undefined to variable, during execution. This forbids use of the *execute* primitive (⍎), as well as the use of *system functions* (such as ⎕ex) operating on the APL name space. *Quad input* is not supported at present. The restrictions on *execute* and *quad input* could be eased somewhat, to permit them to operate on arguments that do not refer to objects in the APL name space, but we consider this to be a low-priority task.

The rank and type of the arguments and results of all APL operations contained in the compilation unit must be computable at compile time, by data flow analysis or by explicit declarations supplied by the user.

The code generated by APEX is non-interactive, in the sense that it is not possible to interrupt execution, edit APL functions, modify variables, and then restart execution, as can be done in an interpreter. [1]

5.2 Design Restrictions

Design restrictions are those imposed by our choice of tools and approach to compilation. In particular, the use of SISAL as an intermediate language imposes several restrictions on the APL language that would not be present if the code generator was retargeted to produce C. Hence, most of these design restrictions could be lifted.

Empty arrays of rank greater than one that contain leading zeros in their shape vectors are not supported. As this is an artifact of SISAL's vectors-of-vectors design, it affects the code generator only. If SISAL were extended to support empty arrays in the APL manner, this restriction would be removed with no changes to APEX.

The APEX compiler does not and, indeed, cannot, support the automatic type promotion capability of APL. For example, the APL expression `+/⍳N` may produce a result of type integer or type double-real, depending on the value of `N`. APEX, by contrast, will only generate a result of the same type as `N`. This restriction arises from two causes: the APEX requirement that the result type of all functions be known at compile time, and the failure of C to provide a method for detecting integer overflow in arithmetic operations.

The compiler produces a result of type double-real for the *floor* and *ceiling* functions operating on double-real arguments. This is at variance with APL, where they may produce integer or double-real results, depending on the magnitude of the largest element of the result.

APEX places a restriction on two rarely exploited language features that permit a function to generate different rank results depending on the value or shape of an argument. The restriction arises from the APEX requirement that result ranks must be computable at compile time. First, the left argument to dyadic transpose must be a compile-time constant. Second, the treatment of singletons in dyadic scalar functions differs from that of APL. Specifically, non-scalar singleton extension is forbidden. As an example of the problem arising from support for arbitrary singletons in dyadic scalar functions, consider the expression `(1 1⍴2)+⍳N`. The APL expression produces a matrix result if `N=1` and a vector result otherwise.

A desirable feature of APEX is that it can detect most *value errors* at compile time, rather than

[1] This is not strictly true, in the sense that debuggers for both SISAL and C could be used in an interactive mode with code generated by APEX, to a limited degree.

during execution. However, as noted in Section 3.2.4, a *value error* arising from a zero-trip `:for` loop will produce an erroneous result, rather than detecting the *value error*. This restriction, arising from SISAL semantics, could be lifted at the cost of some additional loop overhead in the cases where such a situation could arise.

5.3 Implementation Restrictions

Implementation restrictions are those imposed by lack of human resources to write code fragments for the code generator and to solve problems that we, in the interest of expediency, side-stepped. They do not reflect, in any sense, fundamental limitations on the APEX dialect of APL.

Niladic functions are not supported. This restriction is an artifact of the development history of APEX that could be lifted with a small amount of effort.

The APL *goto* (`→label`) is not currently supported; line labels are ignored. Flow control is achieved by use of APL+Win control structures. We intend to provide support for *goto* by adding a compiler phase that implements the algorithms of Erosa and Hendron to map *goto* statements into structured controls [EH93].

System variables currently have fixed values, with the exception that the `⎕io` system variable may be defined to be either zero or one at compile time. Semi-global analysis considerably eases the task of providing proper support for system variables, but we have not had the luxury of time to write that support.

APEX-generated code does not yet support native input-output capabilities. Currently, calls to C sub-functions must be used to achieve this.

A number of APL primitive functions are not supported yet. For example, *roll*, *deal*, *domino*, and the more abstruse *circular functions* are not written yet. We expect to support most of these by calls to C library functions or by compiling APL models of the functions. For example, the Jenkins' model of *domino* could serve as the basis for that primitive. A virtue of this approach is that the performance of such library routines would improve as APEX generated code efficiency improves.

Some functions are only supported on arrays of rank 3 or less.

The APL *bracket axis* notation is not supported. Even though all its capabilities are achievable by use of the *rank* conjunction, it is our intent to support this notation in order to facilitate compilation of legacy applications.

5.4 ISO Standard Compliance

The ISO APL Standard requires that "... a *conforming-implementation* shall provide a reference document that satisfies the following requirements for the documentation of *optional-facilities*, *implementation-defined-facilities*, and *consistent-extensions*." This section constitutes that document.

APEX does not support any of the *optional-facilities* described by the Extended ISO APL Standard [EM93].

APEX *implementation-defined-facilities* are defined as follows. APEX supports Boolean, integer, double-precision floating point, and character data. These data are represented internally in a form that depends on the characteristics of the target system's C compiler. Boolean and character data, as defined by the C characters *char*; integer data defined by the C integers, *int*; double-precision data, as defined by the C real numbers, *double*. The *atomic vector* used to map members of the *required-character-set* is that of the 7-bit ASCII alphabet. Other elements of the *required-character-set* are mapped in a fortuitous order. There is no guarantee, at present, that the mapping of non-ASCII characters is one-to-one. The *implementation-algorithms*, *implementation parameters*, *internal-value-set*, and *event-types* are not documented yet.

The APEX compiler supports a number of *consistent-extensions* to ISO Standard APL. We introduced several of these extensions as a way to investigate improved performance of generated code. Others were introduced because they were used in our application benchmarks. We mention them briefly here and suggest that the interested reader consult relevant documentation for more detailed information about them.

Rank conjunction This was introduced to let us evaluate its effect on performance. It made a substantial improvement in several application benchmarks by reducing the amount of array copying performed [Ber87, EM93].

Replicate This extension of the APL *compress* derived function to integer left arguments was required by several application benchmarks [Ber80].

From This functional form of *indexed reference* was introduced as a potential method of speeding up the **nmo** application benchmark [Ive96].

Composition Composition of an array with a function was one of the approaches we took to improve the performance of the **mconv** benchmark [HIMW90].

Cut The *cut* of SHARP APL and J, implemented as a variant of the Extended APL *dyadic reduce*, was introduced to improve the performance of the **logd3** and **mconvred** benchmarks [HIMW90, Bat95].

Function assignment This was introduced as part of an attempt to improve the performance of **mconv**.

LCM and GCD These are consistent extensions of the dyadic scalar functions *and* and *or*. They extend the domain of the Boolean functions to integer and double-real arguments. They were introduced in order to support a data base benchmark that we dropped from our suite because of its excessive dependence on vendor-specific extensions.

Bibliography

[Bat95] John K. Bates. Some observations on using Ching's APL-to-C translator. *ACM SIGAPL Quote Quad*, 25(3), March 1995.

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

[Ber80] Robert Bernecky. *Replication*. Technical Report SHARP APL Technical Note 34, I.P. Sharp Associates Limited, September 1980.

[Ber84] Robert Bernecky. Function arrays. *ACM SIGAPL Quote Quad*, 14(4), June 1984.

[Ber87] Robert Bernecky. An introduction to function rank. In *APL88 Conference Proceedings*, pages 39–43, ACM SIGAPL Quote Quad, December 1987.

[Ber93] Robert Bernecky. Array morphology. In Elena M. Anzalone, editor, *APL93 Conference Proceedings*, pages 6–16, ACM SIGAPL Quote Quad, August 1993.

[Ber97] Robert Bernecky. *APEX: The APL Parallel Executor*. Master's thesis, University of Toronto, 1997.

[BH91] Robert Bernecky and R.K.W. Hui. Gerunds and representations. *ACM SIGAPL Quote Quad*, 21(4), July 1991.

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

[CFR*89] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. An efficient method for computing static single assignment form. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 23–35, January 1989.

[Chi86] Wai-Mee Ching. An APL/370 compiler and some performance comparisons with APL interpreter and FORTRAN. In *APL86 Conference Proceedings*, pages 143–147, ACM SIGAPL Quote Quad, July 1986.

[CNS89] Wai-Mee Ching, Rick Nelson, and Nungjane Shi. An empirical study of the performance of the APL370 compiler. In *APL89 Conference Proceedings*, pages 87–93, ACM SIGAPL Quote Quad, August 1989.

[Dic95] Leroy J. Dickey. ASCII transliteration schemes. *ACM SIGAPL Quote Quad*, 26(2), December 1995.

[EH93] Ana M. Erosa and Laurie J. Hendren. *Taming Control Flow: A Structured Approach to Eliminating Goto Statements*. ACAPS Technical Memo 76, McGill University School of Computer Science, 3480 University Street, Montreal, Canada H3A 2A7, September 1993.

[EM93] *Programming Language APL, Extended*. International Standards Organization, ISO N93.03 committee draft 1 edition, 1993.

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

[Ive62] Kenneth E. Iverson. *A Programming Language*. John Wiley & Sons, Inc., 1962.

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

[JO86] Graham C. Driscoll Jr. and D.L. Orth. Compiling APL: the Yorktown APL translator. *IBM Journal of Research and Development*, 30(6):583 – 593, 1986.

[Mor84] *International Standard for Programming Language APL*. International Standards Organization, ISO N8485 edition, 1984.

[Smi79] Bob Smith. A programming technique for non-rectangular data. In *APL79 Conference Proceedings*, pages 362–367, ACM SIGAPL Quote Quad, June 1979.

[WJ92] David B. Wortman and Michael D. Junkin. A concurrent compiler for modula-2+. In *Proceedings of the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation*, pages 68–81, June 1992.

[Wol92] Michael Wolfe. *High Performance Compilers*. Oregon Graduate Institute, 1992. Lecture notes for IBM Canada Ltd.

[WS81] Zvi Weiss and Harry J. Saal. Compile time syntax analysis of APL programs. In *APL81 Conference Proceedings*, pages 313–320, ACM SIGAPL Quote Quad, October 1981.