

THE POLARIS INTERNAL REPRESENTATION

BY

KEITH AARON FAIGIN

B.A., Williams College, 1992

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1994

Urbana, Illinois

ABSTRACT

The Polaris Program Manipulation System is a production quality tool for source-to-source transformations and complex analysis of Fortran code. In this paper we describe the motivations for and the design of Polaris' internal representation. The internal representation is composed of a basic abstract syntax tree on top of which exist many layers of functionality. This functionality allows complex operations on the data structure as well as allowing it to emulate other internal representations. Further, the internal representation is designed to enforce the consistency of the state of the internal structure in terms of both the correctness of the data structure and the correctness of the Fortran code being manipulated. In addition, operations on the internal representation result in the automatic updating of affected data structures such as flow information.

We describe how the system's philosophies developed from its predecessor, the Delta prototyping system, and how they were implemented in Polaris' internal representation. We also provide a number of examples of using the Polaris system.

ACKNOWLEDGEMENTS

I would like to offer my sincerest thanks to my thesis advisor, David Padua, for his substantial role in the work involved in this thesis. I would also like to thank my co-authors of the paper [3], of the same name as this thesis, from which this thesis was merely a slight extension. These co-authors are Jay Hoeflinger, David Padua, Paul Petersen and Stephen Weatheford.

TABLE OF CONTENTS

CHAPTER	PAGE
1 INTRODUCTION	1
2 DEVELOPMENT OF POLARIS PHILOSOPHIES	4
2.1 Delta	4
2.2 Goals and Philosophies of the IR	6
3 IMPLEMENTATION	10
3.1 C++	10
3.2 Support Structures	12
4 CLASSES OF THE IR	15
4.1 Program Class	15
4.2 ProgramUnit Class	15
4.3 Statement Class	17
4.4 StmtList Class	22
4.5 Expression Class	24
4.6 Symbol Class	26
4.7 Symtab Class	27
5 SAMPLE TRANSFORMATION CODE	28
5.1 Simple Loop Distribution	28
5.2 Code Instrumentation	31
5.3 Loop Normalization	33
6 INTER-COMPILER COMMUNICATION	39
6.1 Delta	40
6.2 Communication with Delta	41
7 CONCLUSIONS	44
APPENDIX A: CLASS METHODS	46
REFERENCES	53

LIST OF TABLES

A.1: Many of the methods defined for the ProgramUnit class.	47
A.2: Many of the methods defined for the StmtList class.	48
A.3: Many of the methods defined for all Statement classes.	49
A.4: Many of the methods defined for derived Statement classes.	50
A.5: Many of the methods defined for all Expressions classes	51
A.6: Many of the methods defined for derived Expression classes.	52

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

The goal of the Polaris system is to provide a new parallelizing compiler that is able to efficiently parallelize Fortran programs for a variety of machines, including massively parallel systems and parallel workstations [6]. Polaris is based on our past experiences with the Cedar Fortran project [2]. This project showed us that real programs can be parallelized efficiently and that the techniques needed to achieve good performance are natural extensions of technology available in current parallelizing compilers. Therefore, we decided to use a traditional internal structure for our new compiler enhanced with some features that make it easy to extend and experiment with transformation techniques. This allows us to capitalize on our previous experiences with the KAP/Cedar parallelizing compiler and the Delta program manipulation system (Delta) [7].

The implementation of Polaris is based on Delta which was created as an “open experimental laboratory” [8] in which to prototype, develop and test new source-to-source transformations for Fortran 77 parallelizing compilers. While Delta succeeded in providing an excellent research environment, it was not practical as a production compiler.

Our experience with Delta taught us that many of the features found in the prototyping paradigm are quite valuable. However, the ideal compiler for source-to-source transformations, we believe, would combine the strengths of a prototyping system (its “usability”) with the strengths of a production system (its computational power). Polaris was designed with this in mind.

This paper presents a description of Polaris’ internal representation (IR). We consider the IR to be more than just the structure of the data within the compiler. We also view it as the operations associated with this data structure. Intelligent functionality can frequently go a long way towards replacing complex data structures and it is usually more extensible. Thus, we have chosen to implement the data-side of the IR in the traditional, straightforward form of an abstract syntax tree. On top of this simple structure, however, we can build layers of functionality which allow the IR to emulate more complex forms. Specifically, such forms could include the constructs we found most useful in Delta and the language we used, SETL [9].

Delta, as an open system, provided the user with complete access to the internal representation. This was because the SETL implementation we used did not have a good data-abstraction mechanism. Allowing users full access to the IR frequently resulted in the failure to properly maintain the internal structure, which hindered program development. However, in Polaris, access to the internal representation is controlled through a data-abstraction mechanism. Operations built onto the IR are defined such that the programmer is prevented from violating the

structure or leaving it in an incorrect state at any point in a transformation. We chose to implement Polaris in the object-oriented language C++ as it allowed us both structural flexibility and gave us the desired data-abstraction mechanisms.¹

Another aspect of the functionality of the IR—and another reason why we chose a relatively simple IR structure—is the ability to work with other compiler systems. Through an intermediate communication language, Polaris can capitalize on the strengths of other systems, such as Delta and KAP [4].

Polaris has been used, so far, to implement passes for array privatization, induction variable substitution, forward substitution and inlining. Also, we are close to the completion of FORBOL which is a C++ extension built on top of our IR which allows complex pattern matching within Polaris.

The rest of this paper is organized as follows: in chapter 2 we describe our goals for Polaris and the general philosophies we employed in its design. In chapter 3 we present a description of how these notions were actually implemented in the internal representation. The major classes used on our IR are discussed in chapter 4. We then, in chapter 5, explore some simple examples which demonstrate the use of Polaris. In chapter 6, Inter-compiler communication is discussed and more detail of the Delta system is given.

¹Another object-oriented transformation system is the Sage++ system [1]. In some respects there are similarities between Sage++ and Polaris but there are also differences in terms of both the overall approach and the implementation.

CHAPTER 2

DEVELOPMENT OF POLARIS PHILOSOPHIES

2.1 Delta

The design of Polaris' IR was heavily influenced by its predecessor, Delta, which is implemented in the language SETL (a very high level language with broad support for tuples, sets and maps as intrinsic data types). Delta's IR is based loosely on the VDL[5] and is represented using labeled arcs. Statements, expressions and symbols are each given distinct labels and stored in maps. Each of these structures are, themselves, maps of sub-structures. Delta is described in more detail in chapter 6.

The major strengths of Delta could be summarized as follows

- Programming environment flexibility — two separate versions of the system are available (with fully automatic tools for translating between the two)
 - an interpreted system good for debugging with small program inputs for experimentation and for incremental changes

- a somewhat faster, compiled version capable of dealing effectively with much larger program inputs
- Data structure flexibility — Because the data structures can be changed dynamically, simple changes or additions to a structure often require no change in other modules since the new information can be safely ignored by these modules (if they see it at all). For instance, a new field could be added to the end of a data dependence tuple or to a map without needing to recompile or in any other way affect the workings of modules which do not access this extra information.

The Delta system also had a number of serious weaknesses.

- Because of the flexibility of the data structures, it is possible to add a new field or new data at one point in the structure but to forget to process it elsewhere. No compiler help is available to tell the programmer whether data-structure additions require changes to the functionality elsewhere in the system. This eventually resulted in prototyping system users each evolving their own forms of the system which were not completely compatible with one another.
- Since the users have complete access to the data-structures, it is up to the user to ensure they are kept in a consistent state. This proved to be a major difficulty made even more complicated by the continually changing state of the internal structure.
- Delta lacks robustness. Many programming errors are only caught at run-time because of incorrectly formed data structures.

- Much of the code is not very modular. It is easy for various programmers to write separate utility functions which repeat functionality since no good modularity¹ can be created for each data structure and its basic manipulations.
- It is difficult to keep documentation up to date since the data structures are flexible or “soft.”

Delta’s programming environment, itself, has proven to be the system’s biggest weakness. One of the biggest problems with the environment is that the run-time system is simply too slow, even with the compiled version of SETL. This is most likely the result of the extensive use of map look-ups (which occur internally in SETL as hash table look-ups). It is not uncommon for a complicated set of transformations on a single large program unit to take an hour or more to complete. This tends to make debugging a tedious chore, since it is practical only on very small test files. This is especially true since only the interpreted version of SETL contains debugging support.

In addition, the environment is too inefficient with memory. Much of the reason for this is an extensive use of strings in the data structures as structure labels.

2.2 Goals and Philosophies of the IR

We wanted our IR to be a very general structure on top of which more complex structures could be emulated. Thus, regardless of what form the IR takes, from the user’s point of view, the IR could seem to be one of nearly any traditional (or non-traditional) representation. This general strategy is complemented by a number of additional philosophies.

¹The language SETL2—the SETL compiler that Delta uses—does, in fact, support modules, but ISETL—the interactive SETL interpreter—does not. In order for Delta to run interactively, as well as in compiled form, we could not use features, such as modules, which are not found in ISETL.

The most pervasive philosophy in Polaris is that of consistency. In response to what proved to be a substantial hurdle in developing transformations in Delta, Polaris was designed to guarantee the correctness of the program representation as much as is efficiently possible. Thus, in general, it should not be possible for the internal structure to be compromised by incorrect transformation code. In addition, the correctness of the Fortran program being manipulated must be maintained. Transformations are, therefore, never allowed to let the code enter a state which is no longer proper Fortran syntax. The system also guarantees the correctness of all flow information. We are also working towards the guarantee that all data-dependence information is kept correct, but these routines are not yet developed to the point where we can determine whether this is actually feasible. This is realized through automatic incremental updates of this information as a transformation proceeds.

We believe that automatic consistency-maintenance will drastically decrease the time required to develop new optimizations within Polaris' production system. Our experience with the Delta system showed us that although greater flexibility and some extra efficiency may be obtained by allowing the internal structure to temporarily fall out of a consistent state, too often the internal structure was not properly restored. This often resulted in incorrect code and time-consuming bugs. We believe that since less flexibility is required in the production system, this approach is merited by the decreased development time.

In addition to maintaining a consistent state, we also require a very robust system. In general, we have tried to detect as many errors as is possible at compile time and, when that was not possible, catch and explain run-time errors. Some of the features which we have implemented in order to realize our goal of robustness—while maintaining consistency—include

- supplying many commonly-needed methods so that users would seldom feel the need to duplicate code or meddle with the system.
- requiring all structures to be fully defined when they are created to avoid the dangers of accidentally “forgetting” needed sub-structures.
- hiding internal structure details which are not necessary for the user to see or alter.
- the strict control over how the IR can be accessed and modified. The Polaris user is only allowed to make incremental changes which keep the system state consistent and correct. For example, statements inserted into a Fortran program are required to be well-formed with respect to multi-statement constructs. For instance, a DO statement cannot be inserted separately from its matching ENDDO statement, since the statement list would enter an incomplete and inconsistent state.
- the detection of aliased structures (structure sharing is not allowed) and the reporting of their existence with a run-time error. For example, it would be an error to create a new expression and insert it into two different statements without first making a copy of the object.
- freeing the programmer from worrying about tedious memory details through the clear indication of ownership of structures and reference counting. The programmer should always be able tell whether he owns a given structure and is, therefore, responsible for its maintenance and deallocation. Further, dangling pointers and their associated problems are avoided through reference counting.

- the detection of the premature destruction of any part of the IR. Data required by the internal representation is protected from accidental deletion.
- extensive error avoidance and checking throughout the system through the liberal use of assertions. Within Polaris, if any condition or system state is assumed, that assumption is specified explicitly in a `p_assert()` (short for “**Polaris** assertion”) statement which checks the assumed condition and reports an error if the assumption is incorrect.

The most important aspect of a prototyping system which we wished to retain in Polaris was its extensibility. Due to the nature of SETL’s built-in map structures, Delta allowed new information to be easily added to its internal representation. Unfortunately, this resulted in many of the problems encountered in trying to maintain the structure’s consistency. We felt it was imperative for the production system to be similarly scalable, but that it be done in a safe manner. As new needs and requirements are discovered, we must be able to safely add additional structures to the IR just as Delta was able to simply add new arcs.

We also required that the IR’s environment allow transformations to be expressed in a simple and straightforward manner. It would not be enough to have a complete set of high-level manipulation methods; we needed them to also have consistent and clear semantics. This includes ideas as simple as rigorous naming conventions as well as more complex concepts such as the indication of structure ownership. Our ultimate goal was to create a system where the development and implementation of algorithms would not be hindered by the internal representation.

CHAPTER 3

IMPLEMENTATION

In this section we describe the implementation of our IR. We begin by describing our motivations for using the language C++ as well as describing how we made use of the features the language provides. This is followed by a discussion of the support structures used.

3.1 C++

We chose to implement Polaris in the object-oriented language C++. The object-oriented paradigm was perfect for supporting the philosophies of the system and C++, specifically, was chosen primarily for its popularity and flexibility.

C++ provides the modularity and efficiency which was lacking in Delta's SETL implementation and, further, provided a superior environment for a team-developed project. C++ was also ideal in that it provided data-hiding mechanisms which allow us to keep tight control over the interface to each structure. We were able to make the complete structure, as well as each sub-structure, an object which could only be accessed through specific methods. Therefore, we

were able to specify all the methods for manipulating the statement list such that any affected structures are updated and we are also able to ensure that the structure has not been violated.

Further, these methods allow needed functionality to be layered on top of the basic structures. Thus, on top of our relatively simple IR, we can emulate more complex structures. Another important benefit of using an object-oriented language is that it provides much of the extensibility which we found so important in Delta. New structures can be added to objects in the IR without affecting the original structures and adding new structures requires very little reprogramming.

C++ also allows the form of all constructors (the routines which instantiate new objects) to be specified. Thus, we are able to ensure that only well-formed and complete objects are created. Further, all destructors (routines called when an object is deleted or falls out of scope) ensure, through reference counting, that relevant parts of the data-structure are not being deleted or are marked invalid and then trapped on reference. In addition, C++ allows reference variables as well as pointers. Throughout the system, a pointer indicates ownership of data, which, in general, means the owner is responsible for its deallocation. A reference variable indicates that the object is owned by another structure and, therefore, must not be deleted.

Many naming conventions are used in the system to promote internal consistency. Of particular importance are those used in conjunction with ownership indication. In order to comply with our ownership conventions, most functions which return an object whose ownership is not being transferred do so by means of a C++ reference. However, in certain instances it must be possible for the function to indicate that the requested object does not exist. In this situation, two corresponding methods are used. The first method has the postfix “`_valid`”

appended to it and indicates whether the requested object exists. The second method has the postfix “`_guarded`” appended to it and returns a reference to the requested object.

For example, in a function call expression, the method `parameters_valid()` returns true if the call has parameters and the method `parameters_guarded()` returns a reference to the parameters. Calling a guarded method which is not valid results in a run-time error. However, reference functions that always succeed do not have a suffix.

Although the above naming convention adheres to our ownership conventions, in some cases it can be rather cumbersome. In a few specific situations we allow functions which do not transfer ownership to return a pointer since it is useful to return a pointer to NULL to indicate that the requested object does not exist. In these situations the postfix “`_ref`” is appended to the function name to indicate that ownership is not being transferred and the object should be treated as though it were a reference.

For example, there is a statement method, `next_ref()`, which allows access to the statement lexically following the given statement. This method should return a reference to the appropriate statement since ownership of the statement is not being transferred. However, rather than using the valid/guarded form, it is much more convenient to return a pointer and indicate that no such statement exists by returning NULL. Thus, the “`_ref`” form is used

In general, C++ provided us with an environment which allowed us to implement our philosophies within Polaris.

3.2 Support Structures

The underlying support system for the IR is just as important as the representation itself. In order to provide full support for the internal representation as well as user code, we have created

an infrastructure of support classes that are heavily used both internally and externally. These structures conform to our conventions, such as ownership indication and naming conventions, and help support many of our philosophies. Further, these structures also make use of the `p_assert()` command for assertion checking as well as perform reference counting.

This infrastructure currently includes a **Collection** class hierarchy which includes lists, sets and a variety of maps. These structures each exist in two forms: ownership and reference. An ownership structure takes control of—and responsibility for—all objects which are inserted into it. Ownership structures insure, through reference counting, that, for instance, objects are not prematurely deleted and that memory is properly deallocated when an object is deleted. Once an object has been placed in an ownership collection, the collection is responsible for its maintenance. An object can only be “owned” by one collection. If a collection is required to contain elements already owned by other structures, a reference structure is used. Reference structures do not take ownership of objects and, in fact, require that inserted objects are already owned.

An example of the use of these structures can be seen in the representation of statements. The statements of a program are kept in an ownership list (**List**). If this list were deleted, the memory used by each statement would be freed. Each statement also contains information on the set of statements which are reachable in the flow-graph. In this case a reference set (**RefSet**) is used. Deleting the statement which contains this set—which would also delete the set—would not affect the statements contained in the reference set. If, however, a statement were deleted which was referenced in the **RefSet**, the statement would be marked invalid and any attempt to reference it from the **RefSet** would result in a run-time error (a `p_assert()`

would be tripped). Since an object can only be owned by single **Collection**, our policy of disallowing structure-aliasing is automatically enforced.

There also exists an **Iterator** class for iterating through any of the collection or reference collection classes. Since each of the collection classes are derived from a base **Collection** class, some functionality is common to all of them. The iterators take advantage of this common functionality. The benefit of this is that an iterator declared as **Iterator<Statement>** could be used to iterate over any collection of statements such as **List<Statement>** or **Set<Statement>** as well as any reference collection such as **RefList<Statement>** or **RefSet<Statement>**.

A specific kind of iterator, called a **KeyIterator**, can be used for iterating over map structures. For example, the structure **Map<Symbol, Statement>**, which is a collection of tuples representing maps of **Symbols** onto **Statements**, can be traversed by an iterator of the form **KeyIterator<Symbol, Statement>**. **KeyIterators** differ from **Iterators** in that they provide methods for accessing the key of a map (in this case, the **Symbol**) as well as the data as the iterator traverses the map structure.

Essentially all of the classes used in Polaris are derived from the class **Listable** which contributes information necessary to indicate ownership of an object and allows the object to be placed in any collection. However, all of the **Collection** classes, including the reference collections, are class templates. We rely heavily on templates for compile-time type checking. For example, a type-error would result from trying to insert an **Expression** into a **RefList<Symbols>**. A similar error would result from trying to traverse a **List<Statement>** with an **Iterator<Expression>**. Without templates, this compile-time error detection would not be possible.

CHAPTER 4

CLASSES OF THE IR

In this section we describe each of the major classes used in the IR in detail. We begin with the basic program class in the first section. In subsections which follow we describe the classes used for representing program units, statements, statement lists, expressions, symbols and symbol tables.

4.1 **Program Class**

The **Program** class is nothing more than a collection of **ProgramUnits**. Methods are included for reading complete Fortran codes as well as displaying them. There are also methods for adding additional **ProgramUnits** as well as merging **Programs**.

4.2 **ProgramUnit Class**

The **ProgramUnit** class is mostly a holder for the various data structure elements which make up a Fortran program unit. This form is, essentially, an abstract syntax tree. A **ProgramUnit** may be of any of the following types:

- **BLOCK_DATA_PU_TYPE** — a `BLOCK DATA` program unit
- **PROGRAM_PU_TYPE** — a main program
- **SUBROUTINE_PU_TYPE** — a subroutine
- **FUNCTION_PU_TYPE** — an external function

The type of a **ProgramUnit** can be determined using the method `pu_class()`.

The **ProgramUnit** class contains and allows access to its component data structures, which are instances of the following classes:

- **StmtList** — a list of all executable program unit statements, if any
- **Symtab** — a symbol table of all symbols used in the program unit
- **DataList** — a list of the information contained in this program unit's `DATA` statements
- **CommonBlockDict** — a dictionary of all common blocks referenced by this program unit
- **EquivalenceDict** — a dictionary of this program unit's variable equivalence classes
- **FormatDict** — a dictionary of this program unit's `FORMAT` statement information
- **WorkspaceStack** — a stack of temporary data structures associated with this program unit which the user can define and use for a specific transformation pass. These structures will remain with the program unit until the pass has completed, unaffected by other transformation passes.

In addition to functions for accessing these data structures inside a **ProgramUnit** object, there are methods for such operations as

- printing or displaying the program unit on any C++ stream (either in Fortran format or with moderate or extensive debugging information)
- copying entire **ProgramUnit** objects
- translating **ProgramUnit** objects to and from the intermediate language format for conversion between the Polaris internal representation and other compiler systems, such as Delta.
- managing the **WorkSpaceStack**. This includes accessing data allocated by the current transformation pass as well as deallocating all structures created by a given pass.

The form of many of the methods can be seen in Table A.1 in the appendix. We discuss some of the more important class structures contained in the **ProgramUnit** class in the following sections.

4.3 Statement Class

We have chosen to implement statements as simple, non-recursive structures kept in a simple statement list (which is described in more detail in the next section). Thus, we have not implemented statement blocks directly. However, we have made the implementation flexible enough that methods which simulate the existence of statement blocks can easily be implemented on top of the current **Statement** class. Furthermore, other more complex structures could be emulated on top of this basic structure, such as control dependence graphs.

Statements are implemented by an abstract base statement class which contains the structures common to all statements. For each specific type of Fortran statement, a distinct class is derived from the base class which contains additional structures specific to that statement.

This class hierarchy allows modifications of and additions to specific statements to be kept local to the statement. In addition, however, if a new method is needed for all statement types, it needs to be implemented only in the base class.

All of the fields declared in the base class (and which, therefore, exist in all statements) are accessed through public methods. Among these fields are

- sets of successor and predecessor flow links which are implemented in the form of reference sets of statements.
- sets of memory references. These include **in_refs**, **out_refs** and **act_refs** which are respectively memory reads, writes and actual parameters accessed by the statement.
- an **outer** link which points to the innermost enclosing DO loop or is null if there is no enclosing DO loop.
- a **WorkspaceStack** of temporary data associated with the **Statement** which used for a specific transformation pass.

Whenever practical, we have implemented the methods such that any modification to a statement results in the updating of affected data, in order to retain consistency. Tables A.3 and A.4, in the appendix, specify many of the methods available to the statement classes.

Each derived statement class may declare additional fields. Among the most common fields declared by derived statement classes are the **follow** and **lead** fields. Since the statement list is implemented as a singly-nested structure, compound statement types, such as block-IFs, are implemented with multiple statements much like they are expressed in Fortran syntax. Thus, a full block-IF (without an ELSEIF clause) is represented in the IR by an **IfStmt** class object

followed in the statement list by some number of statements delimited by an **ElseStmt** which is itself followed by statements delimited by an **EndIfStmt**. The **follow** and **lead** fields connect the statements of these compound structures. For instance, the **follow** field found in an **IfStmt** would point to the next unit of the block-IF which could be either an **ElseStmt**, an **ElseIfStmt** or an **EndIfStmt**. The **lead** field points in the ‘other direction’.

The **DoStmt** declares a number of fields in addition to those declared by the base statement type. The **follow** field within a **DoStmt** points to its corresponding **EndDoStmt** (the DO-CONTINUE construct is not supported and is automatically converted to DO-ENDDO form by the parser), and, likewise, the **follow** field of an **EndDoStmt** points to the corresponding **DoStmt**. In addition, fields are declared which specify the index of the loop as well as the initial, limit and step expression. Each of these fields is an **Expression** tree.

Another important method declared in the base class (but redefined by each derived class) returns an iterator which traverses the expressions contained in that statement. This iterator may traverse 0 expressions, as in an **EndDoStmt** statement, or up to 4, as exist in a **DoStmt**. This method, along with similar methods in the expression class, make it quite easy to, for instance, traverse all the expressions in a loop body statement by statement.

In order to increase the robustness of the structure, all methods which access data fields are declared within the base statement class and are overridden in the derived classes which use them. For example, the methods which access the ‘step’ field are only applicable to the **DoStmt** but are declared in the base class. The base class definition of the **step()** method, like all other base method definitions, calls an error-routine while the redefinition in **DoStmt** performs the specified operation. With this scheme, if a method is called for a statement to

which it is not applicable, a Polaris error will be reported and the system can either try to continue or can perform a controlled abort.

Although this design has the disadvantage of moving the detection of some errors from compile-time to run-time, it has two hopefully larger advantages. The first is that this method generally decreases the time required to compile routines developed using the production system, due to the way C++ compiles large systems. Specifically, the user, in general, only needs to include the header file of the base **Statement** class and not those of the classes derived for particular statements, since all of the methods needed are already defined in the base class; this reduces the compile time of user programs, which in turn makes the debugging process easier¹.

The second reason has to do with the fact that the **StmtList** class contains a list of references to the base **Statement** class. By the C++ rules of typing, it is legal for a reference or pointer to a base class to actually point to a derived class, and this capability is used extensively throughout our system. While iterating through a list of **Statements**, for example, the programmer will receive a reference to the base **Statement** class. Once he has determined the type of **Statement** that reference refers to, he would normally have to then typecast the reference into the correct derived class in order to be able to access the methods appropriate to that statement type. However, we believe that the large number and variety of typecasts required by such a system creates an unnecessarily large possibility for errors made by programmers typecasting to the wrong class type. (These types of errors are especially easy to make when changes are made to the system or to the user program.) Such errors can neither be detected nor controlled by a C++ compiler or by the run-time system itself, and can be extremely difficult

¹The single exception to this rule is that if the user needs to create new statements rather than just modifying current ones, that user must include the appropriate derived class header files in order to access the constructors for that class. Generally only a few such header files, if any, need to be included by a user program.

to trace. However, by placing all possible methods directly into a base class, we gain complete run-time detection and control of errors of this type. The cost of this technique, unfortunately, is an abundance of virtual methods.

In order to ensure that no incomplete structures can exist within the program, the constructors for statements require all fields needed to completely define the statement. For example, the **DoStmt** constructor requires the statement tag—a unique string which identifies the statement which is used primarily for debugging—as well as expressions for the index, initial value, limit and step of the loop. Exceptions are made to this rule for optional structures, such as the optional argument in a return statement (and in these cases the methods to access them are in “**_valid**” and “**_guarded**” forms).

As a simple example, the DO statement header

```
DO I = 1, 10, 2
```

can be created (given a **ProgramUnit**) with the following constructor call, which gives the new **DoStmt** a tag of “S10”:

```
ProgramUnit pgm;
...
Statement *stmt = new DoStmt("S10", id("I", pgm),
                             constant(1), constant(10), constant(2));
```

The call `id("I", pgm)` does a search for the symbol `I` in `pgm`’s symbol table and returns an **IDExpr** expression referring to the symbol. The `constant(data)` call creates a new constant expression of the appropriate type. In this case an integer constant expression (**IntConstExpr**) object with the specified integer value is returned.

4.4 StmtList Class

The **StmtList** class is derived from the class template **List<Statement>**. The **StmtList** class, however, overrides many of the basic list operations to include automatic updating of the flow graph whenever any statement or block of statements is deleted, inserted or moved. Currently, the information automatically updated includes the set of memory references, control flow information and loop-nesting information. We are also working towards allowing data-dependence information to be updated automatically, but the routines are not developed to the point where we can determine whether this is efficient.

In addition to this basic functionality expansion, additional operations are available, including:

- returning an iterator over selected parts of the statement list such as the body of a DO loop, all statements of a specific type or the entire program.
- finding a statement by its tag
- copying, deleting, unlinking or moving any well-formed sublist of statements
- inserting any single statement or any well-formed list of statements
- inserting specific multi-block statement groups, such as a block-IF statement framework or a DO-ENDDO group.
- print all statements in the list to any C++ stream in either Fortran form or in a debugging form which displays flow information, memory references and other internal fields.

These and many of the other available methods are specified in Table A.2, in the Appendix.

To maintain complete control of consistency inside the **StmtList** class, the insertion, deletion, unlinking, moving and copying of statements or statement lists are all given a number of restrictions. The first of these is that the block to be processed must be entirely well-formed with regard to multi-block statements such as DO loops and block-IF statements. This restriction is checked at run-time. (At the same time, the **follow** links, flow graph and other internal structures are automatically updated.) In addition, some further restrictions are placed. For example, deleting a block containing a statement which is referenced by another statement outside of the statement block being deleted is flagged at run-time as an error.

Because of these restrictions, it is not possible, for instance, to sequentially insert a DO statement, followed by the statements inside the DO loop, followed separately by the ENDDO statement. Instead, there are two options which provide plenty of flexibility to the programmer. The first is to call one of the several intrinsic methods of **StmtList** to create an empty DO loop (i.e. a header and an ENDDO), and then to singly insert the statements of the body separately in-between these two delimiter statements. The second method is to create a **List<Statement>** statement list (which has no restrictions whatsoever on the order or type of insertions), and then to insert the entire **List<Statement>** into the **StmtList** at once. The syntax of the new list of statements is checked as the list is inserted.

We have attempted to make the insertion, deletion, unlinking, copying and moving of statements within a **StmtList** robust against errors and dangling pointers.

As a simple example of the use of a **StmtList** object, consider the following short C++ code which iterates through all of the assignment statements in a **StmtList** and prints them (by default with debugging information) to the standard output:

```
StmtList stmt_list;
```

```

...
for (Iterator<Statement> stmt_iter = stmt_list.stmts_of_type(ASSIGNMENT_STMT);
     stmt_iter.valid();
     ++stmt_iter)
{
    cout << stmt_iter.current();
}

```

Notice that the `stmt_iter.valid()` expression returns true if the `stmt_iter` iterator is valid. That is, if there are still statements over which to iterate, and the `++stmt_iter` statement causes `stmt_iter` to update its current pointer to the next applicable statement.

4.5 Expression Class

Expressions are represented by a tree structure. They are implemented in much the same way as statements, in that an abstract base **Expression** class declares structures common to all expressions and specific expressions are derived from the base. However, most expressions inherit from three intermediate derived classes: unary expressions (**UnaryExpr** class), binary expressions (**BinaryExpr** class) and non-binary expressions (**NonBinaryExpr** class). These are used to represent expressions with one, two and possibly more than two sub-expressions, respectively. For example, a `.NOT.` expression is represented with a unary node; and subtraction, division and `.EQ.` are represented with a binary node. The non-binary class represents expressions which can have an unlimited number of arguments. This is used mostly for operators which are assumed to be commutative and associative by our symbolic simplification routines, such as addition, multiplication, and several logical operators. The non-binary class is also used to represent lists of expressions, such as the list of actual or formal parameters to a procedure.

Other expression classes are derived which describe specific expression types such as identifier expressions (**IDExpr** class) and integer constant expressions (**IntConstExpr** class). Also, many expressions are derived from **UnaryExpr**, **BinaryExpr** and **NonBinaryExpr** for the sole purpose of defining methods with more readable names for accessing the sub-expressions. For instance, since the **FunctionCallExpr** class is derived from **BinaryExpr**, from which it inherits the functions `left()` and `right()` to access its two subexpressions, which are respectively the function being called (represented by an **IDExpr**) and the parameter list. However, instead of requiring the user to abide by this somewhat ambiguous notation, two new methods named `function()` and `parameters()` are added to the **FunctionCallExpr** class to make the accesses to these fields clear and self-documenting. The `parameters` expression, as well as a number of other cases where lists of sub-expressions are needed, are represented by a **CommaExpr**. In addition to parameter-lists, **CommaExpr** trees are also used to represent the list of subscripts in an array reference.

The base **Expression** class includes fields which specify the expression as well as type information. A type includes the Fortran data type (integer, real, etc..) and the size, making types such as “INTEGER*8” and “INTEGER*4” both possible and distinguishable. In addition, fields are declared which are used for expression simplification. Finally, each **Expression** class also has a method for traversing over all sub-expressions, much like we saw in the **Statement** class.

Another example of an expression which derives from the **BinaryExpr** class is the expression to represent array references, **ArrayRefExpr**. The **BinaryExpr** class declares its two fields which are then accessed through **ArrayRefExpr**’s methods `array()` and `subscript()`. One of the benefits of having a binary expression class is that methods which are applicable

to all expressions with two sub-expressions can be defined there and will be inherited by all such expressions. Thus, in addition to simply contributing two fields to an array reference, the binary expression also contributes to the **ArrayRefExpr** class inherited methods which check whether the expression has any side-effects, as well as numerous methods which help in such operations as expression simplification.

All of the safeguards which were implemented within the **Statement** class are also implemented here. This includes the declaration of default methods at the base level which call error routines. However, unlike the **statement** class, constructors are not available to the programmer. In place of the constructors, expressions are created through a complete set of functions provided by the **Expression** class. These functions were designed to provide the user with a simpler means of creating expressions. Frequently, these functions perform additional tasks in creating the desired expression, such as determining the correct type based on the expression's sub-expressions. Also, since the functions only create expressions on the heap, the programmer is protected from mistakingly allocating these dynamic objects statically.

Many of the methods available to the **Expression** classes are enumerated in Tables A.5 and A.6, in the appendix.

4.6 Symbol Class

The Symbol class hierarchy is set up in a very similar manner to that of the Expression and Statement class hierarchies. The abstract class Symbol defines all possible functions for the derived classes, and the leaves of the Symbol class hierarchy correspond to the different types of symbols possible in a program unit. Six such symbol types are currently defined, represented by

the **BlockDataSymbol**, **FunctionSymbol**, **ProgramSymbol**, **SubroutineSymbol**, **SymbolicConstantSymbol** and **VariableSymbol** classes. All such objects may be inserted into the **Symtab** class (see below). As with all classes, all of the required fields may generally be given directly to the constructor. For example, to create a **VariableSymbol** to represent the Fortran variable **XY_ARRAY** defined in the Fortran lines

```
DOUBLE PRECISION XY_ARRAY(0:100, -50:50)
SAVE XY_ARRAY
```

one could use the following C++ code:

```
Symbol *new_symbol = new VariableSymbol(
    "XY_ARRAY",
    make_type(DOUBLE_PRECISION_TYPE),
    NOT_FORMAL,
    IS_SAVED,
    new ArrayBounds(constant(0), constant(100)),
    new ArrayBounds(constant(-50), constant(50)));
```

Of course, it is also possible to create assumed-size arrays.

4.7 Symtab Class

The **Symtab** class is our implementation of a symbol table. Its major component is a dictionary of **Symbol** class objects. It provides methods for, among other things, inserting new symbols (with automatic renaming, if desired, in the case of name conflicts), deleting or unlinking symbols, renaming symbols, finding symbols by name, printing all the Fortran lines necessary for specifying all the symbols, and creating an iterator to iterate over every symbol in the symbol table.

CHAPTER 5

SAMPLE TRANSFORMATION CODE

Traditionally, only very brief examples would be given in a paper describing an IR. However, since one of Polaris' greatest strengths is its “programmability” arising from the expressiveness of the IR, we will present a few longer examples of programming transformations in Polaris. Although these examples are still fairly simplistic, they should demonstrate the “feel” of Polaris programming.

5.1 Simple Loop Distribution

We begin with a trivial routine which simply distributes a loop into two loops. The procedure accepts the **StmtList** to be transformed, the loop to be distributed and a reference to the statement which indicates where the loop should be split.

```
// Distribute the loop 'do_loop' such that the first loop
// contains the loop statements up to, but not including
// loop_bound, and the second loop contains the remaining
// statements.

void distribute_loop(StmtList & stmts, Statement & do_loop,
                    Statement & loop_bound)
```

```

{
    p_assert(do_loop.type() == DO_STMT,
        "distribute_loop( ): the statement to be distributed is not"
        "a DO statement.");

    // Pull out statements which belong in the second loop
    List<Statement>      *second_block =
        stmts.grab(loop_bound, *do_loop.follow_ref()->prev_ref());

    // Insert a second loop after the original
    Statement           &second_do_loop =
        stmts.ins_DO_after(do_loop.index().clone(),
                           do_loop.init().clone(),
                           do_loop.limit().clone(),
                           do_loop.step().clone(),
                           *do_loop.follow_ref());

    // Insert the second block of statements into the second loop
    stmts.ins_after(second_block, second_do_loop);
}

```

The procedure begins with a `p_assert` call. A `p_assert()`, as described earlier, is a Polaris assertion used for catching run-time errors. Here, it insures that the statement to be distributed is, in fact, a **DoStmt**¹. The `grab` call specifies that all statements beginning with `loop_bound` and ending with the statement proceeding the `DO_LOOP`'s "follow" statement (i.e. the matching `ENDDO` statement) should be removed from the program and placed in the list `second_block`. Notice that this routine returns a pointer to the list of statements, as opposed to a reference. This indicates that ownership of the list is being passed so the user function is now responsible for deallocating the list. The `ins_DO_after` method specifies that an empty `DO_LOOP` (both the `DO` as well as the `ENDDO`) specified by the first four expression parameters (the index, initial value, limit, and step, respectively) should be inserted after the `follow`

¹This check could be removed by changing the type of `do_loop` from 'Statement &' to 'DoStmt &', but, as explained earlier, this would lead to excessive type-casting which could produce errors.

statement of `do_loop`, which is the `ENDDO` statement. Note that the call to `follow_ref()` returns a pointer (even though ownership is not being passed) and must be dereferenced. This method returns a reference (since ownership is not being passed) to the new `DO` statement. The `ins_after` method simply inserts the removed statements into the second loop. Notice that `second_block` is being passed as a pointer. This indicates that control of the list is being given to the method. Thus, the method, after inserting the statements into the `StmtList`, is able to delete the empty list.

Consider, for example, the following Fortran code.

```
(S1)  DO 10 I = 1,10,2
(S2)      A(I) = B(I) - C(I)
(S3)      B(I) = I
(S4)  ENDDO
```

If the `distribute_loop` procedure was called with the loop `S1` and a `loop_bound` of `S3`, the result would be:

```
(S1)  DO 10 I = 1,10,2
(S2)      A(I) = B(I) - C(I)
(S4)  ENDDO
(ST5) DO 10 I = 1,10,2
(S3)      B(I) = I
(ST6) ENDDO
```

It is important to note that each method called in the `distribute_loop` procedure guarantees that, upon completion, the program is in a consistent state. Thus, structural information, such as flow information, as well as Fortran syntax, is checked and updated. If an inconsistent state is encountered, an error is raised. For instance, if the same call to `distribute_loop`—also with a `loop_bound` of `S3`—was made on the following code

```

(S1)  DO 10 I = 1,10,2
(S2)      IF (I.LT.5) THEN
(S3)          A(I) = B(I) - C(I)
(S4)      ENDIF
(S5)  ENDDO

```

an error would be raised by the call to `grab` since removing the statements `S3` and `S4` results in incorrect Fortran syntax.

5.2 Code Instrumentation

The following is a slightly more complex example of Polaris programming.

```

//-----
// Insert instrumentation into a program unit:
//
//   Around each outer DO loop in the program unit, insert:
//       CALL START_INTERVAL(#)
//   and
//       CALL END_INTERVAL(#)
//   where # is a unique integer for each loop in the
//   program unit
//
//   Assume for simplicity's sake that there are no jumps out
//   of DO loops
//-----

instrument(ProgramUnit & pgm)
{
    // Capture any p_assert() errors here
    P_ASSERT_HANDLER(0);

    // Create and insert the necessary symbols into the
    // symbol table.

    Symbol &start_interval = pgm.symtab().ins(
        new SubroutineSymbol("START_INTERVAL", IS_EXTERNAL,
                             NOT_INTRINSIC, NOT_FORMAL));

    Symbol &end_interval = pgm.symtab().ins(
        new SubroutineSymbol("END_INTERVAL", IS_EXTERNAL,

```

```

        NOT_INTRINSIC, NOT_FORMAL));

// Iterate over all of the DO statements.

int interval_number = 0;

for (Iterator<Statement> do_stmts =
    pgm.stmts().stmts_of_type(DO_STMT);
    do_stmts.valid();
    ++do_stmts) {

    if (do_stmts.current().outer_ref() == NULL) { // If an outermost loop...

        interval_number++; // Get the next intvl #

        // Insert 'CALL START_INTERVAL( interval_number )'
        // before the current DO statement.
        pgm.stmts().ins_before(
            new CallStmt(pgm.stmts().new_tag(), // Unique stmt tag
                start_interval, // Subr. symbol being called
                comma( // Actual parameter list
                    constant(interval_number))),
            do_stmts.current());

        // Find the matching ENDDO statement
        Statement &end_do = *do_stmts.current().follow_ref();

        // Insert 'CALL END_INTERVAL( interval_number )'
        // after the current ENDDO statement.
        pgm.stmts().ins_after(
            new CallStmt(pgm.stmts().new_tag(), // Unique stmt tag
                end_interval, // Subr. symbol being called
                comma( // Actual parameter list
                    constant(interval_number))),
            end_do);
    }
}

// Print the resulting program unit to standard output
// with debugging information.
cout << pgm << endl << endl;

// Print to standard output as Fortran code
pgm.write(cout);
}

```

This example is fairly straightforward and should be easily understood from its comments. One feature, however, which merits some discussion is the call to `P_ASSERT_HANDLER` in the first line of the routine. If a `p_assert()` fails, Polaris performs some appropriate action, usually resulting in the program being aborted. The `P_ASSERT_HANDLER` call specifies the action which should be taken if a `p_assert` fails. If a failure is encountered, control is returned to the point of the `P_ASSERT_HANDLER` and the action specified by the handler is carried out. The 0 argument specifies that Polaris should abort with a description of the failed assertion. It is also possible to specify the name of a routine to be called to act as a trap-handler. Multiple `P_ASSERT_HANDLER` calls can exist within a single program specifying how errors should be handled at different stages of the program's execution.

5.3 Loop Normalization

Finally, we present an example of simple loop normalization. That is, we will normalize a DO loop to have its lower bound be zero (0) and its step be one (1). This could be represented as transforming the loop

```
DO i = e1, e2, e3
  ... i ...
ENDDO
```

into the form

```
DO i = 0, (e2 - e1)/e3, 1
  ... (i*e3 + e1) ...
ENDDO
```

if the bound expressions $e1$, $e2$ and $e3$ have no side effects, or else into a form with as much precalculation of the loop bounds as necessary. For instance, if $e1$, $e2$ and $e3$ are function calls which may have side effects, the output would be in the form

```
INIT = e1
LIMIT = e2
STEP = e3
DO i = 0, (LIMIT - INIT)/STEP, 1
  ... (i*STEP + INIT) ...
ENDDO
```

In either case we must also make sure to coerce the loop bound expressions $e1$, $e2$ and $e3$ into the same Fortran type as that of the loop index variable before using them in other expressions. For simplicity, we assume that loop index variables are never used outside of the loop which they control.

The code to perform this transformation requires the ability to iterate over statements, as we saw in the previous example, as well as over all expressions contained in a statement. It also requires being able to replace all occurrences of a particular symbol inside of an expression. The subroutine for this transformation follows.

```
void normalize(ProgramUnit &pgm, Statement &do_stmt) {
  // Normalize loop do_stmt to have a lower bound of 0 and a step of 1

  // Get new copies of the DO's init, limit and step expressions,
  // and call them respectively e1, e2, e3
  Expression *e1 = do_stmt.init().clone();
  Expression *e2 = do_stmt.limit().clone();
  Expression *e3 = do_stmt.step().clone();

  // Get a reference to the index variable
  Symbol &index_var = do_stmt.index().symbol();

  // Coerce e1, e2 and e3 to the type of the loop index
  // by applying intrinsic functions to the expressions
  // (only if necessary)
  e1 = coerce(e1, index_var.type(), pgm);
```



```

e2 = coerce(e2, index_var.type(), pgm);
e3 = coerce(e3, index_var.type(), pgm);

// If the bound expressions could have side effects, they must
// be precalculated.
e1 = get_precalc(e1, pgm, do_stmt, PRECALC_IF_SIDE_EFFECTS, "INIT");
e2 = get_precalc(e2, pgm, do_stmt, PRECALC_IF_SIDE_EFFECTS, "LIMIT");
e3 = get_precalc(e3, pgm, do_stmt, PRECALC_IF_SIDE_EFFECTS, "STEP");

// Replace the init expression with the constant 0
do_stmt.init(constant(0));

// Replace limit expression with (e2 - e1) / e3
do_stmt.limit(div(sub(e2, e1), e3));

// Replace the step expression with the constant 1
do_stmt.step(constant(1));

// Now find all occurrences of the use of the index variable
// inside the loop and replace them with the expression
// ((index_variable*e3) + e1)

// First we need to specify the replacement expression
Expression *replacement =
    add(mul(id(index_var), e3->clone()), e1->clone());

// Loop through all statements within the loop body
for (Iterator<Statement> stmts =
    pgm.stmts().iterate_loop(&do_stmt);
    stmts.valid();
    ++stmts) {
    // For all expressions to be iterated over, substitute
    // all references to the index variable
    // with a copy of the expression 'reference'
    substitute_var(stmts.current().iterate_expressions(),
        index_var, *replacement);
}

// We don't need this expression anymore--garbage collect it
delete replacement;
}

```

A number of support routines used in this program example need additional explanations.

- `void substitute_var(iterator, symbol, replacement-expr)` —

searches through all the expressions specified by *iterator* for references to *symbol*. Whenever it finds such a reference, it is replaced by a newly-created copy of *replacement-expr*. Currently in development for the Polaris system are additional, even more powerful, expression pattern-matching and replacement routines.

- `Expression *coerce(expr, type, program-unit)` —

Returns a new expression which has been created by coercing the expression *expr* into the type given by *type*. (Of course, if *expr* is already of the same type as *type*, *expr* is returned unchanged.) The type coercion is achieved by adding a call to an appropriate intrinsic function (for instance, `INT()` or `DBLE()`) with *expr* as its argument. If this intrinsic function does not already exist inside *program-unit*'s symbol table, it is added automatically.

- `get_precalc(expr, program-unit, reference-stmt, precalc-condition, precalc-variable-name)` —

Does a precalculation (if necessary) of an expression and returns a new expression which references this precalculated value. With *precalc-condition* set to `PRECALC_IF_SIDE_EFFECTS`, if *expr* could have side effects (that is, if it contains a call to an external function), this function automatically creates a new variable and assigns this variable the value of the expression *expr*. This assignment takes place in a newly-created assignment statement which is placed in *program-unit* just before the statement *reference-stmt*. Of course, to retain consistency, all flow-information is automatically updated.

The function returns an expression referring to the (possibly precalculated) value of *expr*. This expression will be either the original *expr* expression (if no precalculation was necessary) or a reference to the newly-created variable. The name of the new variable is specified by *precalc-variable-name*, which defaults to PC (for “precalc”) if not specified. If a symbol with the specified name already exists, it is automatically renamed to avoid any conflicts. Although this function seems fairly specific for a built-in utility, we have found it to be useful for many transformations.

Also notice that, in the creation of the replacement expression, the expressions **e1** and **e3** are cloned. This is required because these two expressions have already been inserted into the `do_stmt.limit` field. Trying to directly insert these expressions in the replacement expression (instead of inserting clones) would be caught by the **Collection** hierarchy as an attempt to alias the expressions.

As an example of the output of `normalize()`, consider the following Fortran subroutine

```
SUBROUTINE SUB(INIT, ILIMIT, B)
  EXTERNAL FUNC1, FUNC2
  REAL*4 FUNC1, FUNC2, B
  INTEGER*4 INIT, ILIMIT
  DO I = FUNC1(INIT), ILIMIT, FUNC2(B)
    PRINT *, I
  ENDDO
  RETURN
END
```

After `normalize()` has been applied to the single loop in this Fortran subroutine the following output is obtained:

```
SUBROUTINE SUB(INIT, ILIMIT, B)
```

```
INTRINSIC INT
EXTERNAL FUNC1, FUNC2
REAL*4 FUNC1, FUNC2, B
INTEGER*4 INIT, I, STEP, INT, INITO
INITO = INT(FUNC1(INIT))
STEP = INT(FUNC2(B))
DO I = 0, (ILIMIT-INITO)/STEP, 1
  PRINT *, I * STEP + INITO
ENDDO
RETURN
END
```

Notice that this Fortran subroutine already contained a variable named `INIT`, so the new variable created by `normalize()` was automatically renamed from `INIT` to `INITO` when it was inserted into the symbol table.

CHAPTER 6

INTER-COMPILER COMMUNICATION

We have been describing the Polaris IR as consisting of many layers of functionality on top of a simple data-structure. One aspect of this functionality which we have not yet described is the ability to communicate with other compiler systems. The data in the IR can be translated to and from an intermediate language representation. Using this intermediate form, Polaris can work in conjunction with other systems. Currently, Polaris is able to fully communicate with the Delta prototyping system and we are working towards allowing Polaris to work with KAP, as well. Eventually, we hope to be able to perform transformations using other compilers—communicating through the intermediate language—thereby taking advantage of the strengths of other systems as well as avoiding the cost of the needless duplication of transformation code.

We will give an example of how Polaris communicates with Delta system, but, first, we need to describe Delta in a little more detail.

6.1 Delta

As mentioned earlier, Delta is written in SETL, a language which allows a high-level, and to some extent self-documenting implementation of many compiler-related algorithms. The internal data structures in Delta make extensive use of string-labeled SETL maps and are defined in a way which is both clear and extremely flexible.

A Fortran program unit is represented as a map of program unit components. Maps are sets containing only two-element tuples which represent a mapping from the first element to the second element. Printing the domain of a program unit map will yield something like

```
{"statements", "initial_statement", "final_statement",  
  "expression", "syntab", "loop_info", "depend_count",  
  "routine_type", "depend_type"};
```

so that, for instance, `pgm("statements")` produces the set of statements for the program unit `pgm`, and the expression `pgm("statements")("S11")` produces the statement `S11` from the program unit `pgm`. (The CSRD dialect of SETL allows the abbreviation `@S11@statements[pgm]` for the same expression. This abbreviation possibility makes programs a bit easier to read since expressions such as this are very common when working with Delta.) As an example of a Fortran statement, consider the assignment statement

$$B(I) = C(I) * R$$

which could appear in Delta's program unit representation as the following SETL map¹ (the output order of tuples in the map is arbitrary and unimportant):

¹The integers are indices into an expression table.

```
{["successors", {"S28"}], ["out_refs", {49}],
  ["in_refs", {53, 54, 51, 47}], ["act_refs", {}], ["rhs", 55],
  ["next", "S28"], ["prev", "S10"], ["outer", "S8"], ["lhs", 49],
  ["predecessors", {"S10"}], ["st", "ASSIGNMENT"]}]};
```

You may recognize many of the field names from Polaris.

6.2 Communication with Delta

As described earlier, there are many strengths and weaknesses associated with Delta's open and flexible data-structure. We attempted to address Delta's weaknesses in Polaris—with particular regard to its speed—while retaining as many of its strengths as possible. However, we recognize that Polaris is no substitute for Delta if an open, interactive system is desired for prototyping. When this is the case, but some of the computational power of Polaris is still required, the inter-system communication is invaluable.

Within Polaris, the creation of an interface module is fairly straightforward. Consider the following example:

```
extern "C" int
instrument_code(char **data, int *len)
{
    // Capture any p_assert() errors here
    P_ASSERT_HANDLER(0);

    // Create a BinStr object and initialize it to 'data'
    BinStr bin = *data;

    // Create a ProgramUnit class object from the BinStr
    ProgramUnit pgm("PGM1", bin);

    // Insert instrumentation into pgm
    instrument(pgm);

    // Place the pgm into the global store
    PUTag tag = global_store_ins(pgm);
```

```

    return_handle(tag, data, len);
}

```

This is an example of a Polaris routine which can be called from delta. The routine receives as its parameters a program passed as an array of characters and the array's size. This character array is assigned to a binary string of data or a **BinStr**. The **BinStr** is the simple intermediate form used to represent Fortran programs between Polaris and other systems. This example routine translates the **BinStr** into a Polaris **ProgramUnit** and then passes it to the “instrument” routine. The instrumented code is then placed into the ‘global store’. The global store is a depository of codes specified by unique tags. The command “global_store_ins(pgm)” places pgm in the global store and returns the tag now associated with the program. The command “return_handle” is a macro which sets “data” and “len” to return the program's tag.

The program above could be called from within Delta by calling it as a function. For instance, suppose the user wrote a Delta program which reads a Fortran program from a file, performs a series of transformations on it, also within Delta, and then wanted to use the instrumentation code to insert timing calls. Within Delta, this would appear as

```

pgm := read_program("fortran_program.f");
pgm := transform(pgm);
handle := dynamic_callout("instrument", "instrument.o", pgm);
pgm := retrieve_program(handle);

```

The “dynamic_callout” command calls the “instrument” procedure with the argument “pgm” and stores the result of the function in “handle”. In this case, the result is the unique tag specifying the program in the global store. The next command retrieves the program, specified by its tag, from the global store, translates it and stores it in “pgm”. Using the global store and

the intermediate program representation, Polaris is able to communicate with other compiler systems.

The Polaris routine in the example, again, makes use of the `P_ASSERT_HANDLER` call. This call provides additional support for inter-system communication. The 0 argument specifies, as before, that Polaris should abort if an assertion fails. However, rather than aborting entirely, as before, in this case control would be returned to Delta (which is particularly useful if the prototyping system is being used interactively).

Another important Polaris feature used in inter-system communication is the existence of an “overflow” field within, basically, every structure in the IR. The “overflow” field is usually empty. If, however, data is found in the intermediate language form which has no counter-part in Polaris—for instance if a programmer has added special fields to the Delta data-structure—it is stored in the “overflow” field. Then, if the program is translated back in order to return it to the original system, the data from the “overflow” field will be returned, unchanged.

In this fashion, users can still take advantage of some of Delta’s strengths while working within the Polaris system.

CHAPTER 7

CONCLUSIONS

The Polaris system's internal representation was designed with the belief that a source-to-source transformation system, even a production quality system, should create an environment which is practical but which still stimulates good programming practices. We have tried to create a system which is robust, is rigorous in its maintenance of a correct structure and which still allows transformations to be expressed clearly and easily.

The IR's structure, however, is a relatively simple one. We have only just begun to build different layers of functionality on top of the basic IR to provide more complex operations. It was designed so that it can adapt and expand, incorporating new methods of analysis and new forms of information, and emulating new representations of traditional information.

While Polaris' internal representation is far from revolutionary, in and of itself, we believe that the concepts incorporated in its design are useful and important in the creation of transformation system. An IR cannot be simply described as the layout of data within a computer's memory. It is inseparable from the functions and philosophies which maintain it. We have endeavoured to take one of the most basic of the traditional IR forms and add concepts such

as consistency maintenance and layered functionality to create the heart of a complete and powerful system which allows complex analysis techniques and transformations to be developed quickly and easily.

APPENDIX A

CLASS METHODS

ProgramUnit::	
stmts()	Returns a reference to the statement list
clone()	Returns a copy of this ProgramUnit
pu_tag_ref()	Returns the unique tag identifying the ProgramUnit
pu_class()	Returns the type of the ProgramUnit
routine_name_ref()	Returns the name of ProgramUnit (if applicable)
syntab()	Returns the symbol table
data()	Returns the data from all DATA statements
common_blocks()	Returns the dictionary of common blocks referenced in this ProgramUnit
equivalences()	Returns the dictionary of variable equivalence classes
formats()	Returns the dictionary of all FORMAT statements
overflow_ref()	Returns a dictionary of syntax tree labels of unrecognized structures found in the intermediate language
work_stack()	Returns a reference to the stack of transformation pass-specific structures associated with this ProgramUnit
clean_workspace(pass_tag)	Delete all WorkSpaces designated for the specified pass
display(out_stream)	Display with moderate debugging information
display_debug(out_stream)	Display with all debugging information
write(out_stream)	Display in FORTRAN format

Table A.1: Many of the methods defined for the ProgramUnit class.

StmtList::	
first_ref()	Returns a pointer to the first statement
last_ref()	Returns a pointer to the last statement
prev_ref(stmt)	Returns a pointer to the statement lexically before 'stmt'
next_ref(stmt)	Returns a pointer to the statement lexically after 'stmt'
entries()	Returns the number of statements in the StmtList
find_ref(stmt_tag)	Returns a pointer to the statement with the tag 'stmt_tag'
iterate_entry_points()	Returns an Iterator over all entry points in the StmtList
ins_before(new_stmt, ref_stmt)	Inserts 'new_stmt' before 'ref_stmt'
†	
ins_before(stmt_list, ref_stmt)	Inserts all statements in 'stmt_list' before 'ref_stmt'
†	
ins_IF_ELSE_after(...)	Inserts a (possibly empty) block IF-ELSE-ENDIF around existing statements in the StmtList
ins_IF_after(...)	Inserts a (possibly empty) block IF-ENDIF around existing statements in the StmtList
ins_ELSEIF_after(...)	Appends an (possibly empty) ELSEIF clause to an existing block IF statement.
ins_ELSE_after(...)	Appends an (possibly empty) ELSE clause to an existing block IF statement.
ins_DO_after(...)	Inserts a (possible empty) DO statement after 'ref_stmt'
move_block_before(...) †	Moves a block of statements to before a given statement
move_before(...) †	Moves a statement to before a given statement
del(stmt)	Delete 'stmt'
del(stmt ₁ , stmt ₂)	Deletes all statements from stmt ₁ to stmt ₂
copy(...)	Returns a copy of a block of statements
stmts_of_type(...)	Returns an Iterator over all statements of specified types
iterate_loop_body(do_stmt)	Returns an Iterator over all statements in 'do_stmt's' body
iterator()	Returns an Iterator over the entire StmtList
iterator(stmt ₁ , stmt ₂)	Returns an Iterator over all statements from stmt ₁ to stmt ₂ lexical order
new_tag()	Returns a unique statement tag
new_label()	Returns a new label

Table A.2: Many of the methods defined for the StmtList class. † indicates that there also exists an “_after” form of the method.

Statement::	
clone()	Returns a copy of the statement
stmt_class()	Specifies what kind of statement this is
next_ref()	Returns a pointer to the lexically next statement
prev_ref()	Returns a pointer to the lexically previous statement
succ()	Returns the set of successor statements in the control-flow graph
pred()	Returns the set of predecessor statements in the control-flow graph
in_refs()	Returns the set of variables read by the statement
out_refs()	Returns the set of variables written by the statement
act_refs()	Returns the set of actual parameters accessed by the statement
outer()	Returns the innermost enclosing DO loop
line()	Returns the line number in the source code
overflow_ref()	Returns a dictionary of syntax tree labels of unrecognized structures found in the intermediate language
tag()	Returns a unique tag indentifying this statement
assertions()	Returns the list of assertions associated with the statement
relink_ptrs(program_unit)	Change all identifiers within subexpressions to refer to 'program_unit's' symbol table
work_stack()	Returns the stack of WorkSpaces associated with the statement

Table A.3: Many of the methods defined for all Statement classes. These methods are defined in the base Statement class and are available to all derived statements.

... Stmt::	
lhs() †	Returns a reference to the expression on the left hand side of an AssignmentStmt
rhs() †	Returns a reference to the expression on the right hand side of an AssignmentStmt
follow_ref()	Returns a pointer to the next statement of a compound structure
lead_ref()	Returns a pointer to the previous statement of a compound structure
matching_if_ref()	Returns a pointer to the corresponding IfStmt of an EndIfStmt
matching_endif_ref()	Returns a pointer to the corresponding EndIfStmt of an IfStmt
expr() †	Returns a reference to the expression of a statement with one expression (i.e. IfStmt, ComputedGotoStmt)
index() †	Returns a reference to the the index expression of a DoStmt
init() †	Returns a reference to the the init expression of a DoStmt
limit() †	Returns a reference to the the limit expression of a DoStmt
step() †	Returns a reference to the the step expression of a DoStmt
target_ref() †	Returns a pointer to the target statement of a GotoStmt
label_list()	Returns a reference to the list of targets of a statement with multiple jumps
s_control_guarded()	Returns a reference to the control information list of an I/O statement
s_control_valid()	Returns true if there exists control information in an I/O statement
io_list_guarded() †	Returns a reference to the expressions read and written in an I/O statement
io_list_valid()	Returns true if there are any expressions in an I/O statement
routine_ref() †	Returns a pointer to the symbol of a subroutine call statement or a subroutine entry statement
parameters_guarded() †	Returns a reference to the parameters of a call statement or an entry statement
parameters_valid()	Returns true if there exist any parameters in a call statement or an entry statement

Table A.4: Many of the methods defined for derived Statement classes. These methods are defined in the base Statement class to call error routines and are redefined for the derived classes which use them. † indicates that there exist corresponding methods which insert data into these fields.

Expression::	
clone()	Returns a copy of the expression
op()	Returns the operator of the expression
type()	Returns the Type object of the expression
arg_refs()	Returns a list of references to all of the expression's sub-expressions
arg_list()	Returns the list of the expression's sub-expressions
overflow_ref()	Returns a dictionary of syntax tree labels of unrecognized structures found in the intermediate language
relink_ptrs(program_unit)	Change all identifiers within subexpressions to refer to 'program_unit's' symbol table
is_wildcard()	Returns true if this is an expression used for pattern matching
possible_values()	Returns a list of values this expression may assume
is_side_effect_free()	Returns true if this is known to be free of side-effects
operator==	Compare expressions—also used for pattern matching

Table A.5: Many of the methods defined for all Expressions classes. These methods are defined in the base Expression class and are available to all derived expressions.

...Expr::	
data_ref() †	Returns a pointer to the character data of a string constant expression
value() †	Returns the integer of an integer constant or argument number expression
real_part() †	Returns a reference to the real part of a ComplexExpr
imaginary_part() †	Returns a reference to the imaginary part of a ComplexExpr
array() †	Returns a reference to the array specified in an array reference
subscript() †	Returns a reference to the subscript specified in an array reference
string() †	Returns a reference to the string specified in a SubStringExpr
bound() †	Returns a reference to the bounds specified in a SubStringExpr
left_guarded() †	Returns a reference to the left hand side of a BinaryExpr
left_valid()	Returns true if the left hand side of a BinaryExpr exists
right_guarded() †	Returns a reference to the right hand side of a BinaryExpr
right_valid()	Returns true if the right hand side of a BinaryExpr exists
function()	Returns a reference to the function of a function call
parameters_guarded() †	Returns a reference to the parameters of a function call
parameters_valid()	Returns true if there exist parameters in a function call
expr_guarded() †	Returns a reference to the expression of an UnaryExpr
expr_valid()	Returns true if there exists an expression in an UnaryExpr
iterator symbol() †	Returns a reference to the symbol of an identifier expression

Table A.6: Many of the methods defined for derived Expression classes. These methods are defined in the base Expression class to call error routines and are redefined for the derived classes which use them. † indicates that there exist corresponding methods which insert data into these fields.

REFERENCES

- [1] F. Bodin, P. Beckman, D. Gannon, S. Narayana, and S. Srinivas. Sage++: A Class Library for Building Fortran 90 and C++ Restructuring Tools. Drafter version 0.1.
- [2] Rudolf Eigenmann, Jay Hoeflinger, Greg Jaxon, Zhiyuan Li, and David Padua. Restructuring Fortran Programs for Cedar. *Concurrency: Practice and Experience*, 5(7):553–5737, October 1993.
- [3] Keith A. Faigin, Jay P. Hoeflinger, David A. Padua, Paul M. Petersen, and Stephen A. Weatherford. The polaris internal representation. Submitted for publication.
- [4] Kuck and Inc. Associates. *KAP for SPARC Fortran User's Guide*, beta version 1.0, document 9308006 edition, 1993.
- [5] P. Lucas and K. Walk. On the formal description of pl/i. *Annual Review in Automatic Programming*, 6, Part 3, 1969.
- [6] D. Padua, R. Eigenmann, J. Hoeflinger, P. Petersen, Peng Tu, S. Weatherford, and K. Faigin. Polaris: A new generation parallelizing compiler for mpps. Technical report, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., June, 1993. CSRD Report 808.
- [7] David Padua. The Delta Program Manipulation System. Preliminary Design. Technical report, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., June 1989. CSRD Report 880.
- [8] Paul M. Petersen, Greg P. Jaxon, and David A. Padua. *A Gentle Introduction to Delta*. University of Illinois at Urbana-Champaign, Center for Supercomp. Res. & Dev., June 1992.
- [9] J.T. Schwartz, R.B.K. Dewar, E. Dubinsky, and E. Schonberg. *Programming with Sets: An Introduction to Setl*. Springer-Verlag, 1986.