International Journal of Parallel Programming, Vol. 22, No. 5, 1994

The Polaris Internal Representation[1]

Keith A. Faigin,[2] Stephen A. Weatherford,
Jay P. Hoeflinger, David A. Padua, and Paul M. Petersen

Received November 10, 1993

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 implementation of Polaris' internal representation (IR). The IR 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. Further, the IR is designed to enforce the consistency of the internal structure in terms of both the correctness of the data structures and the correctness of the Fortran code being manipulated. In addition, operations on the IR 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' IR. We also provide a number of examples of using the Polaris system.

KEY WORDS: Internal representation; object-oriented; parallelizing compiler; source-to-source transformation.

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.[1] 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. There-

[1] The research described was supported by Army contract DABT63-92-C-0033. This work is not necessarily representative of the positions or policies of the Army or the Government.

[2] Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, Urbana, Illinois, 61801.

0885-7458/94/1000-0553$07.00/0

fore, 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.[3]

The implementation of Polaris is based on Delta which was created as an "open experimental laboratory"[4] 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.[5]

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 into 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. (Another object-oriented transformation system is the Sage++ system.[6] 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.)

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.[7]

Polaris has been used, so far, to implement passes for array privatization,[8] induction variable substitution, forward substitution, symbolic dependence analysis[9] and inlining. Also, we are close to the completion of FORBOL[10] 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 Section 2 we describe our goals for Polaris and the general philosophies we employed in its design. In Section 3, we present a description of how these notions were actually implemented in the internal representation. In Section 4, we discuss the major classes used in our IR. We then, in Section 5, explore some simple examples which demonstrate the use of Polaris.

2. DEVELOPMENT OF POLARIS PHILOSOPHIES

2.1. 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 nontraditional) representation. This general strategy is complemented by a number of additional philosophies.

The most pervasive philosophy in Polaris is that of consistency. 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 that the control flow information is consistent. 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 at a reasonable cost. Consistency of the control flow graph 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 a 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 member functions (functions used to access and manipulate objects) 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 multistatement 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 (data 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 (deleting a structure which is still being referenced elsewhere in the IR) 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 that we wished to retain in Polaris was its extensibility. In Delta the program was represented as an abstract syntax tree with labelled arcs. Due to the nature of SETL's built-in map structures, Delta allowed new information to be easily added to its internal representation. Additional information was included by simply inserting arcs with unique labels at the appropriate map nodes in the tree. Unfortunately, this resulted in many problems 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 also to have consistent and clear semantics in the form of specific programming guidelines. 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.

3. IMPLEMENTATION

In this section we describe how the philosophies described were implemented in our IR. The general form of our IR is that of an abstract syntax tree implemented in an object oriented fashion such that each node in the tree is an object. This is implemented in the fairly straightforward manner of using classes for statements, expressions, program units, and all other program structures. The more interesting aspect of the IR is the form of these objects and the functionality contained within them.

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, objects which could only be accessed through specific member functions. Therefore, we were able to specify all the member functions 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 member functions 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 to be specified. Thus, we are able to ensure that only well-formed and complete objects are created. Further, all destructors 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, passing a pointer indicates transfer of 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. These exceptions are made explicit through naming conventions. For instance, the postfix "_guarded" indicates that a corresponding member function (with the postfix "_valid") should be queried first to ensure that the requested object exists. Similarly, the postfix "_ref" indicates that although a pointer is returned, ownership is not being passed (and a NULL value should be checked for).

In general, C++ provided us with an environment which allowed us to implement our philosophies witin 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 while they are still being referenced 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 be 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 was 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 a single **Collection**, our policy of disallowing structure-aliasing is automatically enforced. The complete set of classes available in the Collection hierarchy can be found in Table I.

Essentially all of the classes used in Polaris can be placed in a collection. (In order for a class to be placed in a collection, it must derive from the class **Listable** which contributes information necessary to indicate ownership of an object.) 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.

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 subsection. In the

Table I. The Collection Hierarchy[a]

Collection::	
Basic Collections:	
List	List structure which takes ownership of objects
RefList	List of references to objects
Set	Set structure which takes ownership of objects
RefSet	Set of references to objects
Stack	Stack structure which takes ownership of objects
RefStack	Stack of references to objects
Element	Structure for the ownership of a single object
RefElement	Structure to reference a single object
Database	A database is a mapping from keys to data where the keys are distinguished and ordered by some *inherent value*
Database	Implements the mapping from keys to data where the data are owned
RefDatabase	Implements the mapping from keys to data references
Dictionary	Implements the mapping from strings to data which are owned
RefDictionary	Implements the mapping from strings to data references
Map	A Map is a mapping from keys to data where the keys are distinguished and ordered based on *memory address*
Map	Implements the mapping from keys to data which are owned
RefMap	Implements the mapping from keys to data references
Iterator	Class which iterates over any Collection structure
KeyIterator	Similar to Iterator except it includes methods to access the key information of any Map or Database

[a] Each of these is implemented as a template class.

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**. Member Functions are included for reading complete Fortran codes as well as displaying them. There are also member functions 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 a main program, a `BLOCK DATA` program unit, a subroutine, or an external function.

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 member functions for general operations such as copying and printing **ProgramUnit** objects as well as translating them to and from the intermediate language format for conversion between the Polaris internal representation and other compiler systems, such as

Table II. Many of the Methods Defined for the ProgramUnit Class

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)
symtab( )	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(output_stream)	Display ProgramUnit with moderate debugging information
display_debug(output_stream)	Display ProgramUnit with all debugging information
write(output_stream)	Display ProgramUnit in FORTRAN format

Delta. The form of many of the member functions can be seen in Table II. We discuss some of the more important class structures contained in the **ProgramUnit** class in the following subsections.

4.3. Statement Class

We have chosen to implement statements as simple, nonrecursive structures kept in a simple statement list (which is described in more detail in the next subsection). Thus, we have not implemented statement blocks directly. However, we have made the implementation flexible enough so that member functions 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 program dependence graphs.[11]

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 member function 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 member functions. 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 which may be 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 is used for a specific transformation pass.

Table III. Many of the Methods Defined for All Statements. These Methods are Defined in the Base Statement Class and are Available to All Derived Statements

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 possibly read by the statement
out_refs( )	Returns the set of variables possible written by the statement
act_refs( )	Returns the set of actual parameters possibly 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 identifying this statement
assertions( )	Returns the list of assertions associated with the statement
iterate_expressions( )	Return an iterator over the statement's expressions
simplify_expressions( )	Simplify all of the expressions in the statement
pre_directives( )	Return compiler directives placed before the statement
post_directives( )	Return compiler directives placed after 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

Whenever practical, we have implemented the member functions such that any modification to a statement results in the updating of affected data, in order to retain consistency. Tables III and V specify many of the member functions 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 DO-ENDDOs or IF constructs, are implemented with multiple statement objects. A DO-ENDDO,

Table IV. Some of the Methods Defined for the StmtList 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)[a]	Inserts 'new_stmt' before 'ref_stmt'
ins_before(stmt_list, ref_stmt)[a]	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(...)[a]	Moves a block of statements to before a given statement
move_before(...)[a]	Moves a statement to before a given statement
del(stmt)	Delete 'stmt'
del($stmt_1$, $stmt_2$)	Deletes all statements from $stmt_1$ to $stmt_2$
grab(stmt)	Remove 'stmt' from StmtList and return it
grab($stmt_1$, $stmt_2$)	Remove and return statements from $stmt_1$ to $stmt_2$
modify($stmt_{old}$, $stmt_{new}$)	Replace $stmt_{old}$ with $stmt_{new}$
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

[a] Indicates that there also exists an "_after" form of the method.

for instance, consists of a **DoStmt** and an **EndDoStmt** which delimit the statements within the loop. (The redundant DO-CONTINUE construct is not supported and is automatically converted to DO-ENDDO form by the parser.) The **follow** and **lead** fields connect the statements of these compound structures.

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** and, likewise, the **follow** field of an **EndDoStmt**

Table V. 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

...Stmt::	
lhs( )[a]	Returns a reference to the expression on the left hand side of an AssignmentStmt
rhs( )[a]	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( )[a]	Returns a reference to the expression of a statement with one expression (i.e. IfStmt, ComputedGotoStmt)
index( )[a]	Returns a reference to the index expression of a DoStmt
init( )[a]	Returns a reference to the init expression of a DoStmt
limit( )[a]	Returns a reference to the limit expression of a DoStmt
step( )[a]	Returns a reference to the step expression of a DoStmt
target_ref( )[a]	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( )[a]	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( )[a]	Returns a pointer to the symbol of a subroutine call statement or a subroutine entry statement
parameters_guarded( )[a]	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

[a] Indicates that there exist corresponding methods which insert data into these fields.

points to the corresponding **DoStmt**. The index of the loop as well as the initial, limit, and step expression are implemented as **Expression** trees.

Another important member function 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 member function, along with similar ones in the expression class, make it quite easy to, for instance, traverse all the expressions in a loop body.

In order to increase the robustness of the structure, all member functions which access data fields are declared to be virtual within the base statement class and are overriden in the derived classes which use them. For example, the member functions 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()` function, like all other base member function definitions, calls an error-routine while the redefinition in **DoStmt** performs the specified operation. With this scheme, if a member function 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 member functions needed are already defined in the base class; this reduces the compile time of user programs, which in turn makes the debugging process easier. (The 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 in a particular transformation routine.)

The second advantage 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 program will receive a reference to the base **Statement** class. Once it has been determined what type of **Statement** that reference refers to, it would normally be necessary to typecast the reference into the correct derived

class. Only then would it be possible to access the member functions appropriate to that statement type. In a system based on an abstract syntax tree, the vast majority of references are to a base class since the type of each node can vary. 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 a transformation pass.) Such errors can neither be detected nor controlled by a C++ compiler or by the run-time system itself, and can be extremely difficult to trace. However, by placing all possible member functions 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 member functions.

The necessity of type-casting from a base class is not unique to Polaris. The SAGE++ system deals with the situation by providing a global function for every SAGE++ class which accepts a pointer and, if the pointer is of the specified class, returns it cast into the correct class. Otherwise a null value is returned. For example, to determine if a statement, `s`, is a 'for statement', the function `isSgForStmt(s)` (for "is SAGE++ for statement") is used. If `s` is a 'for statement', it is returned cast in to the 'for statement' class, otherwise null is returned. Consider the following examples in SAGE++ (left) and Polaris (right):

```
SgStatement *s = ....
SgForStmt *loop;
if (loop = isSgForStmt(s)) {
      ....
   i = loop->step();
      ....
}
```

```
Statement *s = ....
if (s->stmt_class() == FOR_STMT) {
        ....
    i = s->step();
        ....
}
```

Within the SAGE++ if statement, `loop` is guaranteed to be of the correct type so member functions specific to the 'for statement' class can be accessed. If the function `step()` were replaced with a function not appropriate to a 'for statement', SAGE++ would catch it at compile time while Polaris would not catch it until run-time. However, while Polaris has a large number of virtual member functions, SAGE++ requires a large number of global "is ⋯" functions (one for every class).

4.4. StmtList Class

The **StmtList** class is derived from the collection 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.
- 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.

These and many of the other available member functions are specified in Table IV.

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 retrictions, 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 member functions 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 member function 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 using, for example, the `ins_before(...)` member function. 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 nonbinary expressions (**NonBinaryExpr** class). These are used to represent expressions with one, two, and possibly more than two sub-expressions, respectively. Addition and multiplication are represented by non-binary expressions in order to facilitate symbolic analysis.

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 member functions with more readable names for accessing the sub-expressions. For instance, the **FunctionCallExpr** class is derived from **BinaryExpr**, from which it inherits the functions `left()` and `right()` to access its two subexpressions. These 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 member functions named `function()` and `parameters()` are added to the **FunctionCallExpr** class to make the accesses to these fields clear and self-documenting.

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 member function for traversing over all sub-expressions, much like we saw in the **Statement** class.

All of the safeguards which were implemented within the **Statement** class are also implemented here. This includes the declaration of default member functions 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 expressions, which should be dynamic objects, statically.

Polaris also has very powerful expression structural equality and pattern matching routines, as well as pattern-matching and replacement routines.[10] These are based on an abstract 'Wildcard' class, which is derived from Expression. To perform pattern matching, one simply creates

a pattern expression (an expression which may contain wildcards anywhere in the tree) and compares this pattern to an expression using the equality matching member function. These functions have proven to be both powerful and general.

Many of the member functions available to the **Expression** classes are enumerated in Tables VI and VII.

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. Each of these derived symbol objects may be inserted into the **Symtab** class.

4.7. Symtab Class

The **Symtab** class is our implementation of a symbol table. Its major component is a dictionary (a **Collection** class which maps strings to another class) of **Symbol** class objects. It provides member functions for, among other things, inserting new symbols (with automatic renaming, if desired, in the case of name conflicts), deleting or unlinking symbols,

Table VI. Many of the Methods Defined for All Expressions. These Methods Are Defined in the Base Expression Class and Are Available to All Derived Expressions

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
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 VII. 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

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

[a] Indicates that there exist corresponding methods which insert data into these fields.

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.

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 & cut_point)
{
    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(cut_point, *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**. (This 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.) The grab call specifies that all statements beginning with `cut_point` and ending with the statement preceding 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` 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 `cut_point` 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 `cut_point` 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 outermost 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 carrier 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*. Wherever 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 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 insert these expressions directly 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, ILIMIT, I, STEP, INT, INIT0

INIT0 = INT(FUNC1(INIT))

STEP = INT(FUNC2(B))

DO I = 0, (ILIMIT-INIT0)/STEP, 1

  PRINT *, I * STEP + INIT0

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.

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 communicate fully 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.

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 that is practical but that still stimulates good programming practices. We have tried to create a system that is robust, is rigorous in its maintenance of a correct structure, and that 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.

Although we have only just begun to really use Polaris for serious experimentation, we already believe that, in general, our approach was a successful one. As we expected, development time has been greatly reduced. Polaris programmers report that the usually tedious process of converting a high-level algorithm into code has been greatly simplified to the point where a line of the algorithm is easily represented by a single line of Polaris code. Further, it is very unusual for an operation to be desired which is not already in a predefined method.

One drawback to having such a comprehensive internal representation is that learning to program effectively in Polaris is a lengthy process. Users report, however, that Polaris' advantages are worth the price of a steep learning curve. Another drawback of Polaris is that it is quite memory intensive which is to be expected of such a complex system. In terms of execution time, however, users report that they are very satisfied with the system's efficiency. This can be attributed at least partially to the fact that flags exist which allow the extensive runtime checks to be disabled once the code is believed to be in a stable form.

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 a 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 endeavored 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.

REFERENCES

1. W. Blume, R. Eigenmann, K. Faigin, J. Grout, J. Hoeflinger, D. Padua, P. Petersen, W. Pottenger, L. Rauchwerger, P. Tu, and S. Weatherford, Polaris, A New Generation Parallelizing Compiler for MPPs. *Seventh Annual Workshop on Languages and Compilers for Parallel Computing*, Ithaca, NY 1994. CSRD Report No. 1306, University of Illinois, Urbana-Champaign, Center for Supercomputing Research and Development (June 1993).
2. Rudolf Eigenmann, Jay Hoeflinger, Greg Jaxon, Zhiyuan Li, and David Padua, Restructuring Fortran Programs for Cedar. *Proc. of the 1991 Int'l. Conf. on Parallel Processing* (August 1991).
3. David A. Padua, The Delta Program Manipulation system—Preliminary design. CSRD Report No. 808, University of Illinois, Urbana-Champaign, Center for Supercomputing Research and Development (June 1989).
4. Paul M. Petersen, Greg P. Jaxon, and David A. Padua, A Gentle Introduction to Delta, University of Illinois, Urbana-Champaign, Center for Supercomputing Research and Development (June 1992).
5. J. T. Schwartz, R. B. K. Dewar, E. Dubinsky, and E. Schonberg, *Programming with Sets: An Introduction to Setl*, Springer-Verlag (1986).
6. F. Bodin, P. Beckman, D. Gannon, S. Narayana, and S. Srinivas, SAGE++: A Class Library for Building Fortran 90 and C++ Restructuring Tools—DRAFT 0.1.1993.
7. Kuck and Associates, KAP for SPARC Fortran User's Guide. Beta Version 1.0, Document #9308006 (1993).
8. Peng Tu and David Padua, Automatic Array Privatization. In Utpal Banerjee, David Gelernter, Alex Nicolau, and David Padua, (eds.), *Proc. Sixth Workshop on Languages and Compilers for Parallel Computing, Lectures Notes in Computer Science*, Portland, Oregon, Springer-Verlag **768**:500–521 (August 1993).

9. W. Blume and R. Eigenmann, An overview of symbolic analysis techniques needed for the effective parallelization of the PERFECT benchmarks. *Proc. of the 1995 Int'l. Conf. on Parallel Processing*. CSRD Report No. 1332, University of Illinois, Urbana-Champaignn, Center for Supercomputing, Research and Development (January 1994).
10. Stephen A. Weatherford, High-Level Pattern-matching Extensions to C++ for Fortran Program Manipulation in Polaris. CSRD Report No. 1350, University of Illinois, Urbana-Champaign, Center for Supercomputing Research and Development (May 1994).
11. J. Ferrante, K. Ottenstein and J. Warren, The Program Dependence Graph and Its Use in Optimization. *ACM Trans. on Programming Languages and Systems* **9**(1):319–349 (1987).

Printed in Belgium
Verantwoordelijke uitgever:
Hubert Van Maele
Altenastraat 20 – B-8310 St.-Kruis