Sage++: An Object-Oriented Toolkit and Class Library for Building Fortran and C++ Restructuring Tools *

François Bodin, Irisa, University of Rennes
Peter Beckman, Dennis Gannon, Jacob Gotwals, Srinivas Narayana,
Suresh Srinivas, Beata Winnicka, Department of Computer Science, Indiana University

Abstract

Sage++ is an object oriented toolkit for building program transformation and preprocessing tools. It contains parsers for Fortran 77 with many Fortran 90 extensions, C, and C++, integrated with a C++ class library. The library provides a means to access and restructure the program tree, symbol and type tables, and source-level programmer annotations. Sage++ provides an underlying infrastructure on which all types of program preprocessors can be built, including parallelizing compilers, performance analysis tools, and source code optimizers.

1 Introduction

Designing and building a source-to-source translation system is a very time consuming task. However, such systems are often a prerequisite for many compiler and language extension research projects. Sage++ was designed to be a toolkit for such projects. It provides an integrated set of parsers that transform the source program into an internal representation, which we call a program tree; a library of object-oriented methods that manipulate the program tree and give tool builders complete freedom in restructuring it; and an unparser that generates new source code from the restructured internal form.

Such translation systems are used, for example, in studies of language extensions that permit the explicit representation of parallelism, where they are necessary for the construction of compilers that generate target-specific source code containing calls to run-time systems that support concurrency. Examples include the Fortran-D and Fortran 90D compilers [1], and the pC++ compiler [5]. Source-to-source translation systems are also used to construct compilers that tackle the problem of discovering parallelism in sequential source code. These compilers apply restructuring transformations to generate a version of the program that uses directives or explicit parallel constructs understood by the "native" compiler on the parallel machine. Examples of these include the compilers for Parafrase and Parafrase-II [6], and Superb [2]. Sage++ also makes it possible to build library specific optimizations that can be added as a portable preprocessing stage to the C++ compiler.

Often called preprocessors because they do not generate native machine code, all of these systems use extensive syntactic and semantic analysis and sophisticated global optimization to transform source

*This research is supported by ARPA under contract AF 30602-92-C-0135 and Esprit APPARC

code. In each case, these systems have been based on a layer of compiler technology that has required years for each research group to develop. Sage++ makes this layer available in a generalized form. By starting from Sage++ rather than starting from scratch, researchers in these fields can potentially cut years off their development time.

A second motivation for releasing Sage++ is in response to a frequent request by programmers for access to tools that allow them to go beyond simple macros, and create extensions to C++ or Fortran.

2 Sage++ as a Meta-Tool

Sage++ is designed to be a tool with which other tools can be constructed. For example, scientific library designers can use Sage++ to build tools capable of optimizing the source code that links with the library. As an illustration, consider the problem of optimizing the use of a C++ Matrix library package similar to Lapack++ [4]. The library consists of a C++ classes for Vectors and Matrices, with a special mechanism for describing subarrays and subvectors, and where the standard arithmetic operators have been overloaded:

```
class GenMat;

class Vector{
  public:
  Vector(int n){};
  Vector & operator =(Vector &);
  Vector & operator +(Vector &);
  friend Vector &operator *(float x, Vector &v);
  void vecVecAdd(Vector &left, Vector &right); // this = left*right
  void matVecMul(GenMat &M, Vector &x);        // this = M*x
  void scalarVecMul(float x, Vector &v);       // this = x*v
};

class GenMat{ // general matrix
  public:
  GenMat(int i, int j);
  GenMat & operator()( int , int );
  GenMat & operator()( index &,  index &);
  GenMat & operator =(GenMat &);
  Vector & operator *(Vector &);
};
```

A common problem with such algebraic applications of C++ is managing the high cost of generating temporaries and creating copies, in expressions like

```
GenMat M(100,100);
Vector w(100), x(100), y(100), z(100);
z = x+ M*(x+3.2*y + w);
x = z+w;
y = z+x;
```

A naive compiler will allocate temporaries for each of the operations (in this case eight). Furthermore, the assignment operations may create needless copies of the data (in this case three). The above code fragment can be written to use temporaries much more efficiently; member functions that take two arguments should be used. For example, the expression $x = z+w$ can be written as *x.vecVecAdd(z,w)*. Furthermore, because vector addition does not carry any dependences, we do not have to worry about possible aliases, and expressions like $x = z+x$ can be re-written as *x.vecVecAdd(z,x)*. However, this is not true for Matrix Vector multiply. In this case, if *x.matVecMul(M,z)* is written as

```
for (i = 0; i < size; i++) {
  x[i] = 0.0;
  for (j = 0; j < size; j++)
    x[i] += M(i,j)*z(j);
}
```

then we cannot allow x and z to be aliased and a temporary must be generated.

Taking these observations about aliases into account, it is easy to write a Sage++ preprocessor that will transform the first set of expressions above into the code below.

```
GenMat M(100,100);
Vector w(100),x(100),y(100),z(100);
Vector _T0(100);
z.scalarVecMul(3.2,y);
z.vecVecAdd(x,z);
z.vecVecAdd(z,w);
_T0.matVecMul(M,z);
z.vecVecAdd(x,_T0);
x.vecVecAdd(z,w);
y.vecVecAdd(z,x);
```

Such a preprocessor would scan the original code looking for expressions involving the *Vector* or *GenMat* classes. A strategy that the preprocessor might use is to employ the left hand side of each assignment as a free temporary as long as it is not used on the right hand side of the assignment. Notice that one temporary `_T0` was generated because of the alias problem with matVecMul. A more aggressive optimization could have discovered that y could be used instead, since it is redefined later.

A very different application of Sage++ is in constructing a tool for automatic instrumentation of user code. A team at the University of Oregon has used Sage++ to automatically add instrumentation for tracking class member function calls [9]. They use Sage++ to add an instrumentation object to the definition of each member function to be tracked. When the program is run, invocation of any of the instrumented methods calls the constructor of the instrumentation object, which can log the method's invocation. Upon exit from the method, the instrumentation object's destructor can log that event as well.

Yet another application of Sage++ is in the construction of preprocessors for Fortran programs. The best example is the Fortran-S [3] system designed at the University of Rennes. This system relies on user-level annotations embedded in comments, which are extracted by Sage++, and invoke transformations such as loop vectorization, blocking and interchange, which are all coded using Sage++.

Sage++ is also being used in a project at Argonne National Laboratory to perform automatic differentiation of numerical algorithms written in C. Automatic differentiation is the process of augmenting

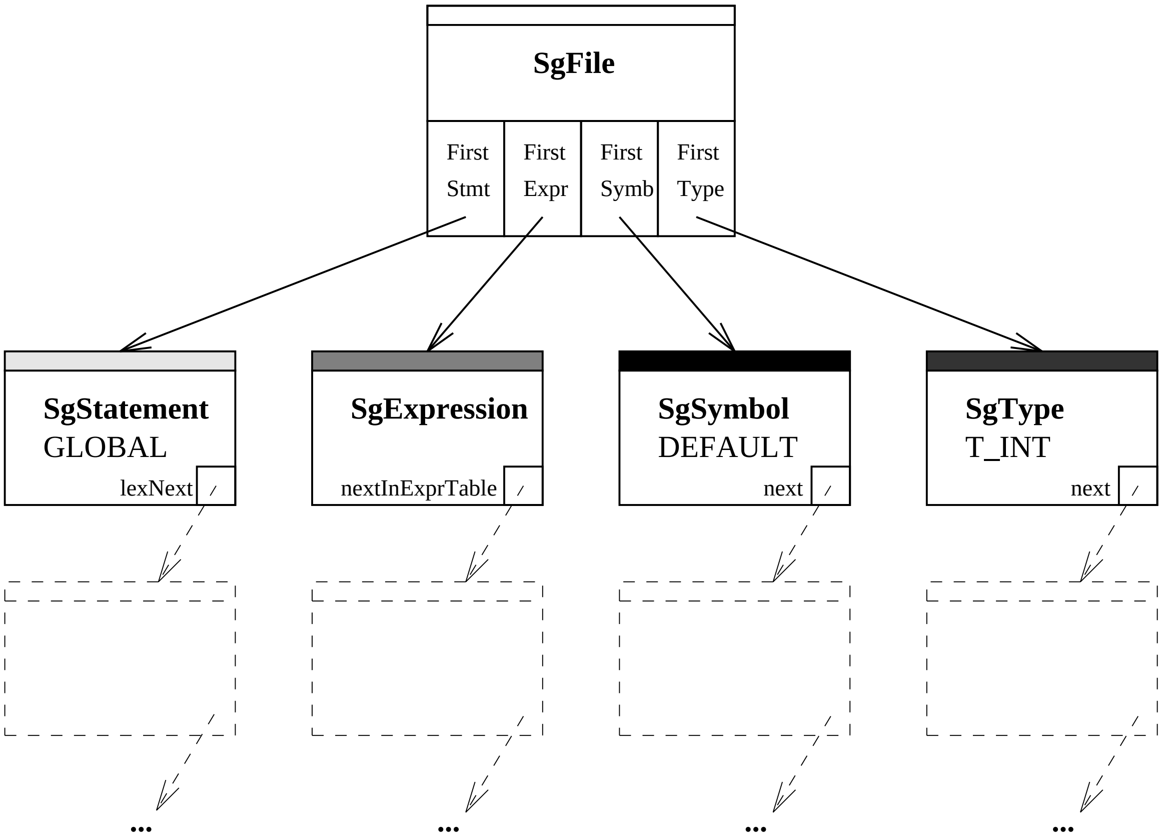

Figure 1: The Four Threaded Trees Linked to *SgFile*

functions in the original code to produce derivative values in addition to the original function outputs [10].

Another Fortran example is a system at the University of Colorado that optimizes certain expressions for parallel execution on the KSR-1. The resulting tool is used by an NSF Grand Challenge project there.

3 Using Sage++

This section provides an overview of the Sage++ library. The library contains four families of classes: project and file classes, corresponding to multi-source application projects and the source files they contain; statement classes, corresponding to the basic source statements in Fortran (with many Fortran 90 extensions), C and C++; expression classes, representing the expressions contained within statements; symbol classes, representing the basic user defined identifiers; and type classes, representing the types associated with each identifier and expression (see Figure 1).

3.1 Parsing

To transform an application program with Sage++, each source file must first be parsed, using either the Fortran parser *f2dep*, or the C/C++ parser *pC++2dep*. The result is a set of machine independent binary files called *.dep* files. Each *.dep* file corresponds to one program tree. *.dep* files are complete translations of the source files they represent, including comments, so they can be used to regenerate the original source. The contents of a *.dep* file can be examined using the *dumpdep* program.

3.2 Project and File Classes

The Sage++ class *SgProject* represents the set of application source files to be transformed. The *SgProject* constructor takes the *.proj* file (a newline-delimited list of *.dep* files) as an argument, and initializes a *SgProject* object by loading each of the *.dep* files:

```
main(){
  SgProject P("MyProject.proj"); // opens and initializes project
  for (int i = 0; i < P.numberOfFiles(); i++){
    printf("Working on file %s...\n", P.fileName(i)");
    // begin transformation of P.file(i) here;
  }
}
```

Each *.dep* file contains a program tree for the code in the source file it represents. The root of the tree in each file is called the global node and its immediate children are the top level definitions and functions in the file. Each *.dep* file also contains a symbol table and a type table; these will be described below.

The class representing files, *SgFile*, provides access to each of these top level definitions and to the symbol and type tables (see Figure 1). *SgFile* member functions can save the restructured program to a standard output stream, string buffer, ASCII file, or *.dep* file. A *.dep* file can be unparsed back into source code by using the *unparse* utility.

3.3 Statements

The Sage++ statement classes are used to represent the basic units of Fortran and C programs. Sage++ has 53 different statement classes divided into five categories: *structured header definitions* like functions, modules and classes; traditional *structured control statements* like do loops and conditionals; *executable statements* like C expressions and subroutine calls; *simple control statements* like goto and return; and *declarations* and miscellaneous Fortran statements like USE, PARAMETER, and I/O statements. Most of the statement class hierarchy is shown in the table at the end of this paper.

The top-level statement class contains member functions that are applicable to all statements. The subclasses add special constructors and access functions. Each statement has several standard attributes, accessible by methods of the top-level class. A source line number, a unique identifier, and a variant tag which identifies the subclass of the statement are attributes of statements. There can also be up to three expressions directly associated with any statement. The C *for* statement is a good example of why three expressions are needed: *for(expr_1; expr_2; expr_3)*{}. Since the basic Sage++ building block is a statement, labels and comments are linked to the preceding statement.

Each statement also has a context, accessible through the top-level statement class: it may have a lexical predecessor and successor, and it may be nested within a control block of another statement or in a block structured definition like a C *struct*. This enclosing statement is called the *control parent* and it defines the basic structure of the program tree.

To illustrate the use of the statement classes consider the following simple example. Suppose we wish to traverse a file and apply an unrolling transformation to all the innermost loops whose bodies consist only of assignment statements. The function is shown below:

```
void UnrollLoops(SgFile *file, int unroll_factor){
  SgStatement *s = file->firstStatement();
  SgForStmt *loop;
  while(s){
    if (loop = isSgForStmt(s)) {
      SgForStmt *inner;
      inner = loop->getInnermostLoop();
      if (inner->isAssignLoop()) inner->unrollLoop(unroll_factor);
    }
    s = s->lexNext(); // the lexical successor of statement s.
  }
}
```

This function illustrates a typical Sage++ program; the main body of the function traverses the statements in lexical order. The variable *s* is a pointer to a generic statement object. There are two ways to tell if a statement is a loop: check to see if the variant is FOR_NODE or use a special casting function, in this case *isSgForStmt()*. A function of the type

```
SgSUBCLASS * isSgSUBCLASS( SgBASECLASS *)
```

is provided for each subclass of *SgStatement*, *SgExpression* and *SgType*. These functions check the variant of the argument. If the object is of the derived type *SgSUBCLASS*, the function returns a pointer to the object, cast to *SgSUBCLASS** (thus permitting access to the special member functions of *SgSUBCLASS*). Otherwise, it returns NULL.

3.4 Expressions

Expressions are represented by trees of objects from the Sage++ expression class. These trees are either degree two (for binary operators) or one (for unary operators).

The base class for expressions, *SgExpression*, contains methods common to all expressions. Every expression may have up to two operands: a left hand side *lhs()* and a right hand side *rhs()*. In addition, each expression has a type and it may have a symbol. The member functions of *SgExpression* allow the programmer to inspect and modify these fields.

In addition, there are special methods that manipulate an entire expression tree whose root is a given node. For example, *replaceSymbolByExpression()* searches an expression for symbol references and replaces them with an expression. To help with building dependence analysis modules, there is a method which will simplify linear algebraic expressions to a normal form and another which will extract the integer coefficient of a symbol in a linear expression.

The hierarchy of Sage++ classes for expressions is shown in the table at the end of this paper. Unlike the classes representing statements, there is not a subclass for every kind of expression. Expression nodes that have their own subclass have a special type of constructor or special fields. The standard binary operators have not been given explicit expression subclasses. Instead, these operators have been overloaded as "friends" of the expression class, so building expressions containing them is a very simple task. For example, to build an expression of the form

$$(X + 3) * Y + 6.4$$

we need two symbol objects and two value expression objects:

```
SgVariableSymb xsymb("X"), ysymb("Y");
SgVarRefExp      x(xsymb), y(ysymb);
SgValueExp       three(3), fltvalue(6.4);
SgExpression &e = (x + three)*y + fltvalue;
```

The variables X and Y are created as symbols, then variable reference expressions to the symbols are created through *SgVarRefExp* objects. Notice that we have not given types to the variables X and Y, and their declarations have not been generated. We will return to that in the next section. In this code, e is now a reference to the root (+) of the program tree for the desired expression(see Figure 2). Also note that the constructors for the value expression class allow any base type value to be created:

```
SgValueExp(int value);
SgValueExp(char char_val);
SgValueExp(float float_val);
SgValueExp(double double_val);
SgValueExp(char *string_val);
SgValueExp(double real, double imaginary);
SgValueExp(SgValueExp &real, SgValueExp &imaginary);
```

To build a C assignment statement of the form

$$X = (X + 3) * Y + 6.4;$$

using the definitions for variables X, Y, *three*, and *fltvalue* given above, we first construct an *SgAssignOp* expression and then use it to build an *SgCExpStmt* object:

```
SgCExpStmt c_stmt(SgAssignOp(x.copy(), (x + three)*y + fltvalue));
```

In Fortran, there are no assignment expressions and we build a statement directly:

```
SgAssignStmt fortran_stmt(x.copy(), (x + three)*y + fltvalue);
```

The expression subclasses provide constructors that make it easy to build expressions and extract values. For example, the constructors for the class *SgArrayRefExp*, used for array references, make building array references simple.

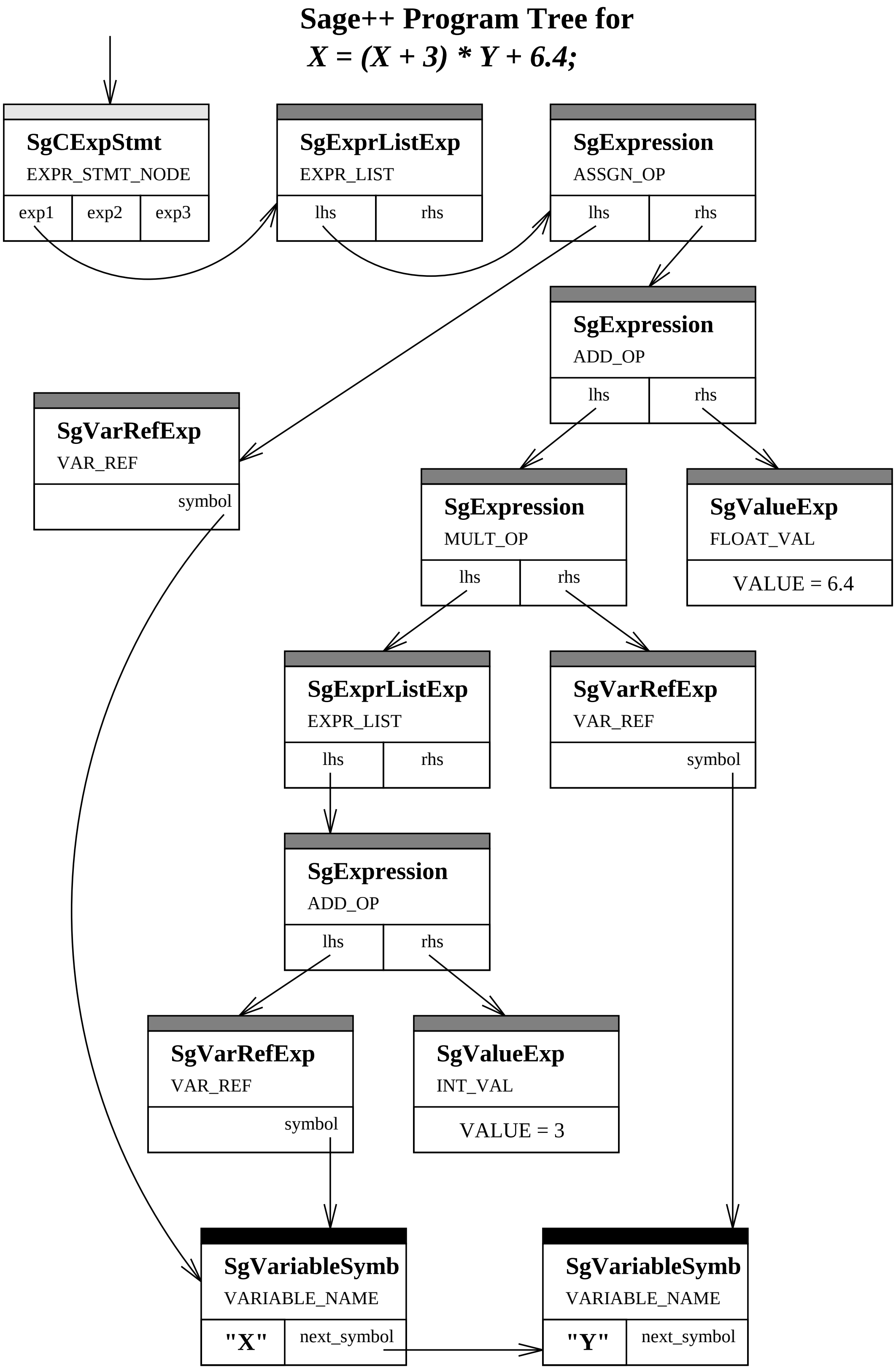

Figure 2: Program Tree Segment

```
SgArrayRefExp(SgSymbol &s, SgExpression &sub1, SgExpression &sub2,
              SgExpression &sub3);
```

builds a 3-D array reference to an array with symbol *s* and three subscript expressions. As another example, vector expressions in Fortran (and our extension to C) are of the form *exp1 : exp2 : exp3* and are represented with the *SgSubscriptExp* class, which has a constructor of the form

```
SgSubscriptExp(SgExpression &lbound, SgExpression &ubound,
               SgExpression &step)
```

as well as access functions to extract the three arguments. So, to build the Fortran 90 array reference `X(1:100:3)` or the "extended" C array reference `X[1:100:3]` we can write

```
SgVariableSymb xsymb("X");
SgValueExp  three(3), one(1), hundred(100);
SgSubscriptExp range(one, hundred, three);
SgArrayRefExp new_expression(xsymb, range);
```

3.5 Symbols and Types

3.5.1 Symbols

The Sage++ base class for symbols is *SgSymbol*. The symbols in each file are organized as a list which is accessible from *SgFile*. Every symbol has a unique *identifier()*, a *type()*, which is described in detail below, and a statement called *scope()*, which is the statement in which the declaration is scoped (i.e. the control parent of the declaration statement). The statement where the variable is declared is given by *declaredInStmt()*.

SgSymbol has three methods for generating copies of a symbol. The simplest method copies only the symbol table entry. Another function copies the symbol and generates new type information, and the third methods copies the declaration body too. For Fortran 90 symbols, the attributes can be inspected or modified.

There are relatively few symbol subclasses:

- **SgVariableSymb** represents basic program variable names. It has methods which return the number of uses of the variable, and the statements and expressions where it is used. Similar functions exist for the definitions of the variable.
- **SgConstantSymb** represents names of program constants. Instances can be constructed by giving an identifier, a scope statement and an expression that defines the value.
- **SgFunctionSymb** represents function and subroutine names. Methods allow access to the symbols in the formal parameter list, the result symbol, and the recursive flag, for Fortran.
- **SgMemberFuncSymb** represents symbols for member functions of classes and structs. The constructor allows specification of the identifier, type, enclosing class and protection status of the function. Methods return the protection status of the function and the symbol table entry of the defining class.

- **SgFieldSymb** This class is used for fields in a C enum statement, or the fields in a struct or class. It has methods similar to those in *SgMemberFuncSymb*.
- **SgClassSymb** represents names of classes, unions, structs, and pC++ *collections* (A data-parallel language extension). Its methods provide access to the fields and functions of the classes it represents.
- **SgTypeSymb** is used for symbols from a C typedef.
- **SgLabelSymb** is for C labels.
- **SgLabelVarSymb** is for Fortran label variables.
- **SgExternalSymb** is for Fortran external functions.
- **SgConstructSymb** is for Fortran construct names.
- **SgInterfaceSymb** is for Fortran 90 module interfaces.

Traversing the symbol table is a very common Sage++ task. For example, consider the problem of looking for the symbol table entry for a given member function in a given class. There are several ways to do this. The code below searches the table for the name of the class. Then it searches the field list for the name of the member function, and finally returns the pointer to the symbol object (if it is found).

```
SgSymbol *findMemberFunction(char *className, char *functionName){
  SgSymbol *s, *fld;
  SgClassSymb *cl;
  SgMemberFuncSymb *f;
  int i;

  for (s=file.firstSymbol(); s ; s = s->next())
    if (!strcmp(s->identifier(), className))
       break;

  if (s == NULL)
     return NULL;

  if (cl = isSgClassSymb(s)){
    for(i = 1, fld = cl->field(i); fld; fld = cl->field(i++)){
      if ((f = isSgMemberFuncSymb(fld)) &&
          !strcmp(f->identifier(), functionName))
         return f;
    }
  }
  return NULL;
}
```

3.5.2 Types

The Sage++ type classes hold basic information about the symbols. As with symbols, Sage++ puts the types from a source file into a list of objects whose head can be accessed through a method of *SgFile*.

The base class for types is *SgType*. Many types are defined in terms of other types. For example, an array type has a base type that may be a pointer type which has a base type that may be an integer or a class, etc. Methods of *SgType* can copy types, and test whether two types are equivalent.

There are eight basic subclasses of *SgType*:

- **SgTypeInt, SgTypeFloat, SgTypeChar, ...** are the basic types.
- **SgArrayType** is used to represent array types. The parser gives each array variable its own array type descriptor object. Member functions can add new dimensions, return the shape and base type of the array, and change the base type.
- **SgClassType** represents types of C structs, Fortran records, C++ classes, C unions, C enums, and pC++ *collections.*
- **SgPointerType** represents pointer types.
- **SgReferenceType** represents C++ reference types, i.e. of the form *int &x*.
- **SgFunctionType** represents types of symbols for functions. A method returns the type of the function's return value; another method allows modification of that type.
- **SgDerivedType** represents the type of C symbols coming from typedef, as well as the type of variables that are of class type.
- **SgDescriptType** is a descriptor object that serves to modify another type object. For example, in the C statement *long volatile int x;* long and volatile are modifiers and int is the base type. The type of *x* is represented by an **SgDescriptType** object that holds the information about the modifiers and the base type.

To illustrate the use of the type table, consider the simple problem of identifying if a variable is of a given user defined class. More specifically, in analyzing the code

```
class myClass;
myClass x, y;
y = x + 2;
```

if *e* is the expression representing the variable reference to *x*, we would like a function *isVarRefOfClass(e, "myClass")* that will return 1 when the class type matches and 0 otherwise. To write this function, we first check to see if *e* is indeed a variable reference. Then, we check to see if the type of the symbol is a derived type. From the derived type we can find the name of the class.

```
int isVarRefOfClass(SgExpression *e, char *className){
  SgSymbol *s;
  SgDerivedType *d;
  SgClassSymb *cl;
  SgVarRefExp *exp;
```

```
  if((exp = isSgVarRefExp(e)) == NULL) return 0;
  s = exp->symbol();
  if((d = isSgDerivedType(s->type())) == NULL) return 0;
  if(cl = isSgClassSymb(d->typeName()))
    if(!strcmp(cl->identifier(), className))
      return 1;
  return 0;
}
```

4 Data Dependence and Data Flow Analysis

Sage++ provides a general framework for data dependence and flow analysis, similar to the one described in [7]. This part of Sage++ has been kept as open as possible to facilitate experimentation. Note that the data dependence and flow analysis routines in Sage++ are still under development, and are subject to occasional modification. The current implementation is limited to Fortran 77[1].

4.1 Data Dependence Analysis

The Omega test [8] is the data dependence test used in Sage++. The data dependence routines in Sage++ provide an interface to the Omega software. Functions are provided to extract data from loops about induction variables, array access in linear form, and data dependences. Data dependence information is provided in the form of distance and direction vectors (the same data dependence information as calculated by Omega). The *depGraph* class stores data dependence information; a subset of that class appears below:

```
class depGraph {
  depNode     *current; // list of dependences
  depNode     *first;
 public:
  SgStatement *loop; // the loop statement
  Set *arrayRef;     // list of array access in the loop in linear form
  Set *induc;        // set of induction variables
  depGraph(SgFile *fi, SgStatement *f,SgStatement *l);
  ~depGraph();
  ...
};
```

The fact that array references are stored in linear form helps to implement the interface to the dependence tests. The dependence graph and other loop related information can be calculated for any function by calling

```
initializeDepAnalysisForFunction(file,function)
```

`depg = new depGraph(file,function,loop)` extracts the dependence graph for the named loop.

[1]The data dependence and flow analysis routines in Sage++ were not part of the initial Sage++ distribution.

4.2 Flow Analysis

Sage++ provides a framework for data flow analysis, to help users write their own flow analysis routines. For example, to implement an iterative forward data flow analysis, the user writes a set of functions to build the *gen* and *kill* sets for each statement, and the function *equal* (which indicates when two elements of a set are equal), and passes those functions to the following Sage++ function:

```
void iterativeForwardFlowAnalysis(SgFile *file,
  SgStatement *func,
  Set *(*giveGenSet)(SgStatement *func,SgStatement *stmt),
  Set *(*giveKillSet)(SgStatement *func,SgStatement *stmt),
  int (*equal)(void *e1, void *e2),...);
```

The class `Set` is provided to help with implementing data sets.

The current version also offers a more general flow analysis framework, with functions such as `controlFlow(SgStatement *stmt,...)`, which returns successors and predecessors of a statement in the control flow graph, and `defUseVar(SgStatement *stmt,...)`, which returns a list of data read and written by a statement. These functions can be used to construct more complex flow analysis tools.

4.3 Loop Transformations

Sage++ provides a set of basic loop transformation tools. The following are some of the loop transformations available:

```
int loopFusion(SgStatement *loop1,SgStatement *loop2)
int loopInterchange(SgStatement *b, int *permut, int n)
int tileLoops(SgStatement *func,SgStatement *b, int *size, int nb)
int distributeLoopSCC(SgStatement *b, int *sccTable,
                      int leadingdim, int numSCC)
...
```

These transformation routines do not check for the validity of the transformations they perform. The conditions required to apply a transformation may be very different from one application of Sage++ to another. In many cases it is known that a transformation is legal, but still the legality cannot be explicitly checked. For example, this is the case when a previous transformation has changed the structure of a loop without updating the data dependence graph. Furthermore, program transformations may also be specified by directives in the code.

5 Finding Out More About Sage++

For complete details about using Sage++, consult the Sage++ User's Guide (about 250 pages). It is the most complete reference on Sage++ available. It includes an introduction and overview of Sage++, a complete description of each class in the library, several example programs, and a complete index.

The Sage++ development team maintains an automated mail server, FTP archive, and a WWW (world wide web) server. To get more information about the Sage++ mailing lists, send a non-empty mail message to:

sage-request@cica.indiana.edu

All the Sage++ papers, manuals, program files, and hypertext documents are also available via the anonymous FTP archive ftp

cica.cica.indiana.edu:pub/sage

A hypertext version of the Sage++ User's Guide and other papers may be accessed through the WWW address

http://www.cica.indiana.edu/sage/home-page.html

Bug reports may be sent to

sage-bugs@cica.indiana.edu

6 Limitations of Sage++

Sage++ has proven to be a powerful tool in our compiler prototyping experiments, but it still has a number of limitations. The most important of these is that it is not easy for users to add language extensions to Fortran or C. To add a new statement to a language, the parser (based on a the GNU Bison version of YACC) must be extended, a new node type must be added to the internal form, and a corresponding subclass added to the Sage++ hierarchy. The *unparser* module, which is table driven, must also be extended to recognize this new node. While we have done this several times (we have added some of the PCF, Fortran-S, Fortran-M and HPF extensions to Fortran and extended C++ to define our pC++ language [5] as well as the proposed CC++ syntax), it is not an easy task, because it requires a complete understanding of the internal parser structures. A future version of Sage++ will work with a different parser generator that will, we hope, simplify this task.

7 Conclusions

The Sage++ object hierarchy provides a flexible and extensible tool for manipulating an internal representation of programs written in C/C++ and Fortran (and its extensions). Coupled with parsers and an unparser, the Sage++ toolkit allows researchers to construct a wide variety of source-to-source transformation systems efficiently, significantly reducing the development time that is usually associated with building such systems from scratch.

Sage++ is still under development. Many of the improvements and extensions already implemented have been suggested by Sage++ users. We would like to thank them all, and in particular Bernd Mohr, Andrew Mauer, Darryl Brown, Michael Golden, and Craig Chase.

References

[1] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu. "Fortran D Language Specification." Tech. Report COMP TR90079 from the Dept. Computer Science, Rice University, March 1991.

[2] P. Brezany, M. Gerndt, V. Sipkova, and H.P. Zima. "SUPERB support for irregular scientific computations." In *Proceedings of the Scalable High Performance Computing Conference (SHPCC-92)*. IEEE Computer Society Press, April 1992, pp. 314–321.

[3] F. Bodin, L. Kervella, and T. Priol. "Fortran-S: A Fortran Interface for Shared Virtual Memory Architectures." In *Proceedings, Supercomputing 93*, Portland Oregon.

[4] J. Dongarra, R. Pozzo, D. Walker, "An Object Oriented Design for High Performance Linear Algebra on Distributed Memory Architectures," In *Proceedings of the First Annual Object-Oriented Numerics Conference (OON-SKI)*, Sunriver, Oregon, Apr. 1993, pp. 257–264.

[5] F. Bodin, D. Gannon, P. Beckman, S. Narayana, S. Yang , "Distributed pC++: Basic Ideas for an Object Parallel Language," In *Proceedings of the First Annual Object-Oriented Numerics Conference (OON-SKI)*, Sunriver, Oregon, Apr. 1993, pp. 1-24.

[6] Polychronopoulos, C., Girkar, M., Haghighat, M., et al, "The Structure of Parafrase-2: An Advanced Parallelizing Compiler for C and Fortran", In *Languages and Compilers for Parallel Computing*, Gelernter, D., Nicolau, A., Padua, D., eds., MIT Press 1990, pp. 423–453.

[7] A. V. Aho and R. Sethi and J. D. Ullman, *Compiler Principles, Techniques, and Tools*, Addison-Wesley, 1986.

[8] W. Pugh, D. Wonnacott "Eliminating False Data Dependences Using the Omega Test" Tech. Report CS-TR-2993, from the Dept. of Computer Science, Univ. of Maryland; an earlier version appeared at the ACM SIGPLAN PLDI'92 conference.

[9] A. Malony, B. Mohr, P. Beckman, D. Gannon, S. Yang, F. Bodin, "Performance Analysis of pC++: A Portable Data-Parallel Programming System for Scalable Parallel Computers", In *Proc. 8th Int. Parallel Processing Symb. (IPPS)*, Cancún, Mexico, Apr. 1994.

[10] A.Griewank, "On Automatic Differentiation", In *Mathematical Programming: Recent Developments and Applications*, M. Iri and K. Tanabe, eds., Kluwer Academic Publishers, 1989, pp. 83–108.

Top Level Block Definition Statements, F=Fortran(90), C=C(++)		
Class Name	Parent Class	meaning
SgProgHedrStmt	SgProcHedrStmt	F: program stmt
SgProcHedrStmt	SgStatement	F: subroutine definition
SgFuncHedrStmt	SgProcHedrStmt	C,F: function definition
SgModuleStmt	SgStatement	F: module stmt
SgInterfaceStmt	SgStatement	F: module interface
SgBlockDataStmt	SgStatement	F: block data stmt
SgClassStmt	SgStatement	C: class{}
SgStructStmt	SgClassStmt	C: struct{}
SgUnionStmt	SgClassStmt	C: union{}
SgEnumStmt	SgClassStmt	C: enum{}
SgBasicBlockStmt	SgStatement	C: { }
Basic Control Statements		
SgForStmt	SgStatement	F: do(), C: for(){}
SgWhileStmt	SgStatement	F: while(), C: while(){}
SgDoWhileStmt	SgWhileStmt	C: do{}while()
SgLogIfStmt	SgStatement	F: if()stmt
SgWhereStmt	SgLogIfStmt	F: where() stmt
SgIfStmt	SgStatement	F: if() then, C: if(){}else{}
SgIfElseIfStmt	SgIfStmt	F: elseif() then
SgArithIfStmt	SgStatement	F: if() less,equal,greater
SgSwitchStmt	SgStatement	F: case, C:switch(){}
SgCaseOptionStmt	SgStatement	C: case x:
Executable Leaf Statements		
SgExecutableStatement	SgStatement	abstract class
SgAssignStmt	SgExecutableStatement	F: assignment
SgPointerAssignStmt	SgAssignStmt	F: pointer <= data
SgCExpStmt	SgExecutableStatement	C: expr, ..., expr;
SgContinueStmt	SgExecutableStatement	F: continue
SgControlEndStmt	SgExecutableStatement	F,C: end of block
SgExitStmt	SgControlEndStmt	F: exit
SgBreakStmt	SgExecutableStatement	C: break
SgCycleStmt	SgExecutableStatement	F: cycle
SgReturnStmt	SgExecutableStatement	F,C: return
SgGotoStmt	SgExecutableStatement	F,C: goto label
SgAssignedGotoStmt	SgLabelListStmt	F: goto variable
SgComputedGotoStmt	SgLabelListStmt	F: goto(x) l1, l2, ...
gCallStmt	SgExecutableStatement	F: call f()
Leaf Declaration Statements		
SgDeclarationStatement	SgStatement	abstract class
SgVarDeclStmt	SgDeclarationStatement	F,C: var decl, C: fnct proto
SgParameterStmt	SgDeclarationStatement	F: parameter()
SgImplicitStmt	SgDeclarationStatement	F: implicit
SgUseStmt	SgDeclarationStatement	F: uses
SgStmtFunctionStmt	SgDeclarationStatement	F: statement function

Expression Classes for C++ and Fortran 90	
Class Name	Meaning
SgExpression	The root class; also home for the basic binary operators: $+, -, *, /, \%, <<, >>, <, >, <=, >=, \&, \vert, \&\&, \vert\vert, +=, \&=,$ $*=, /=, \%=, \hat{}=, <<=, >>=, =, ==, !=, ->, ::, .$
SgValueExp	A base type (i.e. a “literal”) value
SgKeywordValExp	Fortran keyword values in I/O statements, etc.
SgUnaryExp	*expr, & expr, sizeof(expr), −expr, +expr, ++lhd, rhs++, −−lhs, rhs−−, ∼expr, and !expr
SgCastExp	(typename) expr;
SgDeleteExp	delete [size] expr;
SgNewExp	new typename (expr);
SgExprIfExp	(expr1) ? expr2 : expr3;
SgFunctionCallExp	function_name(expr1, expr2, ...);
SgFuncPntrExp	(functionpointer)(expr1,expr2,expr3);
SgExprListExp	Expression lists: expr1, expr2, ... ;
SgRefExp	Fortran name references
SgVarRefExp	Scalar variable references
SgThisExp	C++ ”this” reference
SgArrayRefExp	Fortran X(exp,exp) or C X[exp][exp];
SgPntrArrRefExp	Pointer used as array: (pointer)[index1][index2];
SgPointerDerefExp	*pointer; (also SgUnaryExp)
SgRecordRefExp	StructName.field;
SgStructConstExp	Fortran 90 structure constructor;
SgConstExp	Fortran 90 array constructor;
SgVecConstExp	[exprlist] ;
SgInitListExp	Used for initializations: { expr1,expr2, ... };
SgObjectListExp	Used for EQUIVALENCE, NAMELIST and COMMON statements
SgAttributeExp	Fortran 90 attributes
SgKeywordArgExp	Fortran 90 keyword argument
SgSubscriptExp	vector range op: low:up:stride
SgUseOnlyExp	Fortran 90 USE statement ONLY attribute
SgUseRenameExp	Fortran 90 USE statement renaming
SgSpecPairExp	Fortran default control args to Input/Output stmts
SgIOAccessExp	Fortran index variable bound instantiation
SgTypeExp	Fortran type expression
SgSeqExp	Fortran sequence expression
SgStringLengthExp	Fortran string length expression
SgDefaultExp	Fortran default node
SgLabelRefExp	Fortran label reference