[](https://www.researchgate.net/?enrichId=rgreq-ff733478044614874ea423cdaa0ef001-XXX%26enrichSource=Y292ZXJQYWdlOzI3MjU0MDg7QVM6MTAyMzc3NTQxNjAzMzM1QDE0MDE0MjAxMTY1MDk%3D%26el=1_x_1%26_esc=publicationCoverPdf)

See discussions, stats, and author profiles for this publication at: [https://www.researchgate.net/publication/2725408](https://www.researchgate.net/publication/2725408_Sage_An_Object-Oriented_Toolkit_and_Class_Library_for_Building_Fortran_and_C_Restructuring_Tools?enrichId=rgreq-ff733478044614874ea423cdaa0ef001-XXX&enrichSource=Y292ZXJQYWdlOzI3MjU0MDg7QVM6MTAyMzc3NTQxNjAzMzM1QDE0MDE0MjAxMTY1MDk%3D&el=1_x_2&_esc=publicationCoverPdf)

[Sage++: An Object-Oriented Toolkit and Class Library for Building Fortran and C++ Restructuring Tools](https://www.researchgate.net/publication/2725408_Sage_An_Object-Oriented_Toolkit_and_Class_Library_for_Building_Fortran_and_C_Restructuring_Tools?enrichId=rgreq-ff733478044614874ea423cdaa0ef001-XXX&enrichSource=Y292ZXJQYWdlOzI3MjU0MDg7QVM6MTAyMzc3NTQxNjAzMzM1QDE0MDE0MjAxMTY1MDk%3D&el=1_x_3&_esc=publicationCoverPdf)

**Article** · October 1994

Source: CiteSeer

CITATIONS

107

READS

454

**5 authors**, including:

[](https://www.researchgate.net/profile/Peter-Beckman-2?enrichId=rgreq-ff733478044614874ea423cdaa0ef001-XXX%26enrichSource=Y292ZXJQYWdlOzI3MjU0MDg7QVM6MTAyMzc3NTQxNjAzMzM1QDE0MDE0MjAxMTY1MDk%3D%26el=1_x_4%26_esc=publicationCoverPdf)[Peter H. Beckman](https://www.researchgate.net/profile/Peter-Beckman-2?enrichId=rgreq-ff733478044614874ea423cdaa0ef001-XXX&enrichSource=Y292ZXJQYWdlOzI3MjU0MDg7QVM6MTAyMzc3NTQxNjAzMzM1QDE0MDE0MjAxMTY1MDk%3D&el=1_x_5&_esc=publicationCoverPdf)

[Argonne National Laboratory](https://www.researchgate.net/institution/Argonne_National_Laboratory?enrichId=rgreq-ff733478044614874ea423cdaa0ef001-XXX&enrichSource=Y292ZXJQYWdlOzI3MjU0MDg7QVM6MTAyMzc3NTQxNjAzMzM1QDE0MDE0MjAxMTY1MDk%3D&el=1_x_6&_esc=publicationCoverPdf)

**92** PUBLICATIONS **3,386** CITATIONS

[Dennis Gannon](https://www.researchgate.net/profile/Dennis-Gannon-2?enrichId=rgreq-ff733478044614874ea423cdaa0ef001-XXX&enrichSource=Y292ZXJQYWdlOzI3MjU0MDg7QVM6MTAyMzc3NTQxNjAzMzM1QDE0MDE0MjAxMTY1MDk%3D&el=1_x_5&_esc=publicationCoverPdf) [Microsoft](https://www.researchgate.net/institution/Microsoft?enrichId=rgreq-ff733478044614874ea423cdaa0ef001-XXX&enrichSource=Y292ZXJQYWdlOzI3MjU0MDg7QVM6MTAyMzc3NTQxNjAzMzM1QDE0MDE0MjAxMTY1MDk%3D&el=1_x_6&_esc=publicationCoverPdf)

[](https://www.researchgate.net/profile/Dennis-Gannon-2?enrichId=rgreq-ff733478044614874ea423cdaa0ef001-XXX%26enrichSource=Y292ZXJQYWdlOzI3MjU0MDg7QVM6MTAyMzc3NTQxNjAzMzM1QDE0MDE0MjAxMTY1MDk%3D%26el=1_x_4%26_esc=publicationCoverPdf)**391** PUBLICATIONS **12,243** CITATIONS

[SEE PROFILE](https://www.researchgate.net/profile/Peter-Beckman-2?enrichId=rgreq-ff733478044614874ea423cdaa0ef001-XXX&enrichSource=Y292ZXJQYWdlOzI3MjU0MDg7QVM6MTAyMzc3NTQxNjAzMzM1QDE0MDE0MjAxMTY1MDk%3D&el=1_x_7&_esc=publicationCoverPdf)

[SEE PROFILE](https://www.researchgate.net/profile/Dennis-Gannon-2?enrichId=rgreq-ff733478044614874ea423cdaa0ef001-XXX&enrichSource=Y292ZXJQYWdlOzI3MjU0MDg7QVM6MTAyMzc3NTQxNjAzMzM1QDE0MDE0MjAxMTY1MDk%3D&el=1_x_7&_esc=publicationCoverPdf)

All content following this page was uploaded by [Dennis Gannon](https://www.researchgate.net/profile/Dennis-Gannon-2?enrichId=rgreq-ff733478044614874ea423cdaa0ef001-XXX&enrichSource=Y292ZXJQYWdlOzI3MjU0MDg7QVM6MTAyMzc3NTQxNjAzMzM1QDE0MDE0MjAxMTY1MDk%3D&el=1_x_10&_esc=publicationCoverPdf) on 23 December 2013.

The user has requested enhancement of the downloaded file.

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

Francois 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++ это объектно-ориентированный набор инструментов для создания программных преобразований и инструментов предварительной сборки. Он содержит парсеры для Fortran 77 со многими расширениями Fortran 90, C и C++, интегрированные с библиотекой классов C++. Библиотека предоставляет средства для доступа и реструктуризации дерева программы, таблиц символов и типов, а также аннотаций программиста на уровне исходного кода. Sage++ предоставляет базовую инфраструктуру, на которой могут быть построены все типы программных препроцессоров, включая распараллеливающие компиляторы, инструменты анализа производительности и оптимизаторы исходного кода.

# 1 Введение

Проектирование и создание системы перевода исходного кода в исходный код — очень трудоемкая задача. Однако такие системы часто являются предпосылкой для многих исследовательских проектов по компиляторам и расширениям языка. Sage++ был разработан как инструментарий для таких проектов. Он предоставляет интегрированный набор парсеров, которые преобразуют исходную программу во внутреннее представление, которое мы называем программным деревом; библиотеку объектно-ориентированных методов, которые манипулируют программным деревом и предоставляют разработчикам инструментов полную свободу в его реструктуризации; и депарсер, который генерирует новый исходный код из реструктурированной внутренней формы.

Такие системы трансляции используются, например, в исследованиях языковых расширений, которые допускают явное представление параллелизма, где они необходимы для построения компиляторов, которые генерируют целевой специфичный исходный код, содержащий вызовы к системам времени выполнения, которые поддерживают параллелизм. Примерами являются компиляторы Fortran-D и Fortran 90D [1] и компилятор pC++ [5]. Системы трансляции исходного кода также используются для построения компиляторов, которые решают проблему обнаружения параллелизма в последовательном исходном коде. Эти компиляторы применяют преобразования реструктуризации для генерации версии программы, которая использует директивы или явные параллельные конструкции, понимаемые «родным» компилятором на параллельной машине. Примерами этого являются компиляторы для Parafrase и Parafrase-II [6] и Superb [2]. Sage++ также позволяет строить оптимизации, специфичные для библиотеки, которые могут быть добавлены в качестве переносимого этапа предварительной обработки к компилятору C++.

Часто называемые препроцессорами, поскольку они не генерируют машинный код, все эти системы используют обширный синтаксический и семантический анализ и сложную глобальную оптимизацию для преобразования исходного кода. В каждом случае эти системы были основаны на слое технологии компилятора, на разработку которого каждой исследовательской группе потребовались годы. Sage++ делает этот слой доступным в обобщенной форме. Начиная с Sage++, а не с нуля, исследователи в этих областях могут потенциально сократить годы своего времени разработки.

Второй мотивацией для выпуска Sage++ является ответ на частые запросы программистов на доступ к инструментам, которые позволяют им выходить за рамки простых макросов и создавать расширения для C++ или Fortran.

# 2 Sage++ как Мета - Инструмент

Sage++ разработан как инструмент, с помощью которого можно создавать другие инструменты. Например, разработчики научных библиотек могут использовать Sage++ для создания инструментов, способных оптимизировать исходный код, который связывается с библиотекой. В качестве иллюстрации рассмотрим проблему оптимизации использования пакета библиотеки C++ Matrix, похожего на Lapack++ [4]. Библиотека состоит из классов C++ для векторов и матриц со специальным механизмом для описания подмассивов и подвекторов, и где стандартные арифметические операторы были перегружены:

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 &);

};

Распространенной проблемой таких алгебраических приложений C++ является управление высокой стоимостью создания временных объектов и копий в выражениях типа :

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;

Собственный компилятор выделит временные объекты для каждой из операций (в данном случае восемь). Кроме того, операции присваивания могут создавать ненужные копии данных (в данном случае три). Приведенный выше фрагмент кода можно написать для более эффективного использования временных объектов; следует использовать функции-члены, которые принимают два аргумента. Например, выражение x = z+w можно записать как x.vecVecAdd(z,w). Кроме того, поскольку сложение векторов не несет никаких зависимостей, нам не нужно беспокоиться о возможных псевдонимах, и выражения типа x = z+x можно переписать как x.vecVecAdd(z,x). Однако это не относится к умножению матриц и векторов. В этом случае, если x.matVecMul(M,z) записать как:

for (i = 0; i < size; i++) {

x[i] = 0.0;

for (j = 0; j < size; j++)

x[i] += M(i,j)\*z(j);

}

тогда мы не можем позволить x и z быть псевдонимами, и необходимо сгенерировать временный.

Принимая во внимание эти наблюдения о псевдонимах, легко написать препроцессор Sage++, который преобразует первый набор выражений выше в код ниже.

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);

Такой препроцессор будет сканировать исходный код в поисках выражений, включающих классы Vector или GenMat. Стратегия, которую может использовать препроцессор, заключается в использовании левой части каждого назначения в качестве свободного временного значения, пока оно не используется в правой части назначения. Обратите внимание, что одно временное значение \_T0 было сгенерировано из-за проблемы с псевдонимом с matVecMul. Более агрессивная оптимизация могла бы обнаружить, что вместо этого можно использовать y, поскольку оно переопределяется позже..

Совсем другое применение Sage++ заключается в создании инструмента для автоматического инструментирования пользовательского кода. Группа в Университете Орегона использовала Sage++ для автоматического добавления инструментирования для отслеживания вызовов функций-членов класса [9]. Они используют Sage++ для добавления объекта инструментирования к определению каждой функции-члена, которую нужно отслеживать. Когда программа запущена, вызов любого из инструментированных методов вызывает конструктор объекта инструментирования, который может регистрировать вызов метода. При выходе из метода деструктор объекта инструментирования также может регистрировать это событие.

Еще одно применение Sage++ — создание препроцессоров для программ на языке Fortran. Лучшим примером является система Fortran-S [3], разработанная в Университете Ренна. Эта система опирается на аннотации уровня пользователя, встроенные в комментарии, которые извлекаются Sage++, и вызывают преобразования, такие как векторизация циклов, блокировка и взаимообмен, которые все кодируются с помощью Sage++.

Sage++ также используется в проекте Аргоннской национальной лаборатории для выполнения автоматической дифференциации численных алгоритмов, написанных на языке C. Автоматическая дифференциация — это процесс дополнения функций в исходном коде для получения производных значений в дополнение к исходным выходным данным функции [10].

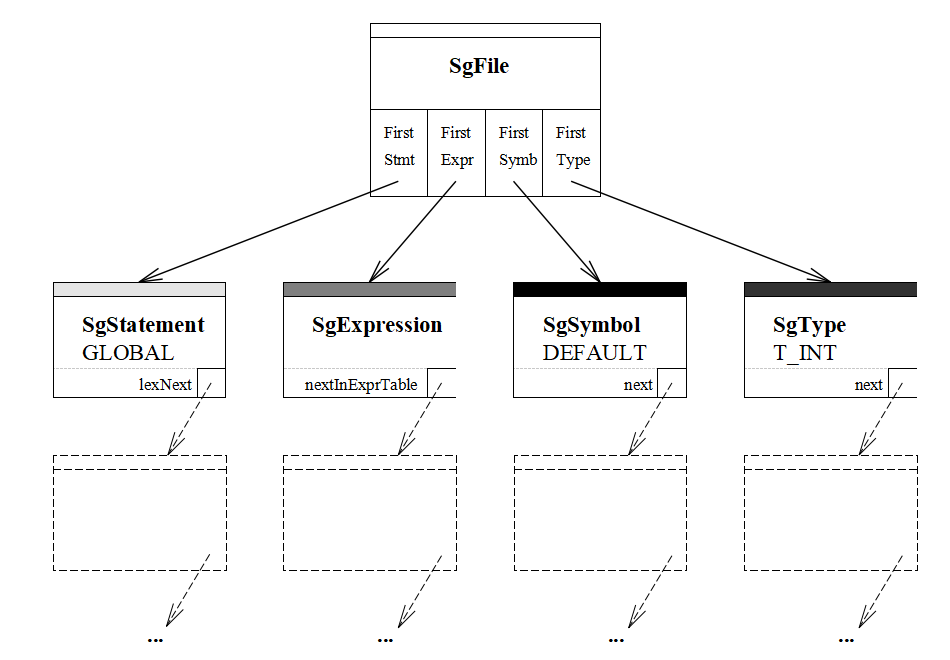


Рисунок 1: Четыре потоковых дерева, связанных с SgFile

Другой пример Fortran — система в Университете Колорадо, которая оптимизирует определенные выражения для параллельного выполнения на KSR-1. Полученный инструмент используется там проектом NSF Grand Challenge.

# 3 Using Sage++

В этом разделе представлен обзор библиотеки Sage++. Библиотека содержит четыре семейства классов: классы проектов и файлов, соответствующие проектам приложений с несколькими источниками и исходным файлам, которые они содержат; классы операторов, соответствующие базовым исходным операторам в Fortran (со многими расширениями Fortran 90), C и C++; классы выражений, представляющие выражения, содержащиеся в операторах; классы символов, представляющие базовые определяемые пользователем идентификаторы; и классы типов, представляющие типы, связанные с каждым идентификатором и выражением (см. рисунок 1).

## 3.1 Parsing

Чтобы преобразовать прикладную программу с помощью Sage++, каждый исходный файл должен быть сначала проанализирован с помощью анализатора Fortran f2dep или анализатора C/C++ pC++2dep. Результатом является набор машинно-независимых двоичных файлов, называемых файлами .dep. Каждый файл .dep соответствует одному программному дереву. Файлы .dep являются полными переводами исходных файлов, которые они представляют, включая комментарии, поэтому их можно использовать для восстановления исходного источника. Содержимое файла .dep можно проверить с помощью программы dumpdep.

## 3.2 Project and File Classes

Класс Sage++ SgProject представляет набор исходных файлов приложения, которые необходимо преобразовать. Конструктор SgProject принимает файл .proj (список файлов .dep, разделенных новой строкой) в качестве аргумента и инициализирует объект SgProject, загружая каждый из файлов .dep:

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;

}

}

Каждый файл .dep содержит дерево программы для кода в исходном файле, который он представляет. Корень дерева в каждом файле называется глобальным узлом, а его непосредственные потомки — определения и функции верхнего уровня в файле. Каждый файл .dep также содержит таблицу символов и таблицу типов; они будут описаны ниже.

Класс, представляющий файлы, SgFile, предоставляет доступ к каждому из этих определений верхнего уровня и к таблицам символов и типов (см. рисунок 1). Функции-члены SgFile могут сохранять реструктурированную программу в стандартный поток вывода, строковый буфер, файл ASCII или файл .dep. Файл .dep можно распарсить обратно в исходный код с помощью утилиты unparse.

## 3.3 Statements

Классы операторов Sage++ используются для представления основных единиц программ Fortran и C. Sage++ имеет 53 различных класса операторов, разделенных на пять категорий: структурированные определения заголовков, такие как функции, модули и классы; традиционные структурированные операторы управления, такие как циклы do и условные операторы; исполняемые операторы, такие как выражения C и вызовы подпрограмм; простые операторы управления, такие как goto и return; и объявления и различные операторы Fortran, такие как USE, PARAMETER и операторы ввода-вывода. Большая часть иерархии классов операторов показана в таблице в конце этой статьи.

Класс операторов верхнего уровня содержит функции-члены, которые применимы ко всем операторам. Подклассы добавляют специальные конструкторы и функции доступа. Каждый оператор имеет несколько стандартных атрибутов, доступных методам класса верхнего уровня. Номер исходной строки, уникальный идентификатор и тег варианта, который идентифицирует подкласс оператора, являются атрибутами операторов. Также может быть до трех выражений, напрямую связанных с любым оператором. Оператор C for является хорошим примером того, почему необходимы три выражения: for(expr 1; ​​expr 2; expr 3)fg. Поскольку базовый строительный блок Sage++ — это оператор, метки и комментарии связаны с предыдущим оператором.

Каждый оператор также имеет контекст, доступный через класс операторов верхнего уровня: он может иметь лексического предшественника и преемника, и он может быть вложен в блок управления другого оператора или в определение блочной структуры, например, структуру C. Этот охватывающий оператор называется родительским элементом управления и определяет базовую структуру дерева программы.

Чтобы проиллюстрировать использование классов операторов, рассмотрим следующий простой пример. Предположим, мы хотим

обойти файл и применить развертывающее преобразование ко всем самым внутренним циклам, тела которых состоят только из операторов присваивания. Функция показана ниже:

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.

}

}

Эта функция иллюстрирует типичную программу Sage++; основная часть функции проходит по операторам в лексическом порядке. Переменная s — указатель на универсальный объект оператора. Есть два способа определить, является ли оператор циклом: проверить, является ли вариант FOR NODE, или использовать специальную функцию приведения, в данном случае isSgForStmt(). Функция типа

SgSUBCLASS \* isSgSUBCLASS( SgBASECLASS \*)

предоставляется для каждого подкласса SgStatement, SgExpression и SgType. Эти функции проверяют вариант аргумента. Если объект имеет производный тип SgSUBCLASS, функция возвращает указатель на объект, приведенный к SgSUBCLASS\* (таким образом, разрешая доступ к специальным функциям-членам SgSUBCLASS). В противном случае она возвращает NULL.

## 3.4 Expressions

Выражения представлены деревьями объектов из класса выражений Sage++. Эти деревья имеют либо степень два (для бинарных операторов), либо степень один (для унарных операторов).

Базовый класс для выражений SgExpression содержит методы, общие для всех выражений. Каждое выражение может иметь до двух операндов: левая часть lhs() и правая часть rhs(). Кроме того, каждое выражение имеет тип и может иметь символ. Функции-члены SgExpression позволяют программисту проверять и изменять эти поля.

Кроме того, существуют специальные методы, которые манипулируют всем деревом выражений, корнем которого является заданный узел. Например, replaceSymbolByExpression() ищет в выражении ссылки на символы и заменяет их выражением. Чтобы помочь в построении модулей анализа зависимостей, существует метод, который упростит линейные алгебраические выражения до нормальной формы, и другой, который извлечет целочисленный коэффициент символа в линейном выражении.

Иерархия классов Sage++ для выражений показана в таблице в конце этой статьи. В отличие от классов, представляющих операторы, для каждого вида выражения нет подкласса. Узлы выражения, имеющие свой собственный подкласс, имеют специальный тип конструктора или специальные поля. Стандартным бинарным операторам не были предоставлены явные подклассы выражений. Вместо этого эти операторы были перегружены как «друзья» класса выражений, поэтому построение выражений, содержащих их, является очень простой задачей. Например, чтобы построить выражение вида:

(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 de nitions for variables X, Y, three, and tvalue given above, we rst 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.

**Sage++ Program Tree for**

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

rhs

lhs

symbol

symbol

next\_symbol

next\_symbol

**"Y"**

**"X"**

**SgVariableSymb**

VARIABLE\_NAME

**SgVariableSymb**

VARIABLE\_NAME

VALUE = 3

**SgValueExp**

INT\_VAL

**SgVarRefExp**

VAR\_REF

**SgVarRefExp**

VAR\_REF

**SgVarRefExp**

VAR\_REF

lhs

**SgExpression**

ADD\_OP

rhs

**SgExprListExp**

EXPR\_LIST

rhs

lhs

**SgExpression**

MULT\_OP

symbol

rhs

lhs

**SgExpression**

ADD\_OP

rhs

lhs

**SgExpression**

ASSGN\_OP

|  |  |  |
| --- | --- | --- |
|  | | |
| **SgCExpStmt**  EXPR\_STMT\_NODE | | |
| exp1 | exp2 | exp3 |

|  |  |
| --- | --- |
|  | |
| **SgExprListExp**  EXPR\_LIST | |
| lhs | rhs |

|  |
| --- |
|  |
| **SgValueExp**  FLOAT\_VAL |
| VALUE = 6.4 |

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 le are organized as a list which is accessible from SgFile. Every symbol has a unique identi er(), 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 modi ed.

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 de nitions of the variable.

SgConstantSymb represents names of program constants. Instances can be constructed by giving an identi er, a scope statement and an expression that de nes 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 ag, for Fortran.

SgMemberFuncSymb represents symbols for member functions of classes and structs. The constructor allows speci cation of the identi er, type, enclosing class and protection status of the function. Methods return the protection status of the function and the symbol table entry of the de ning class.

SgFieldSymb This class is used for elds in a C enum statement, or the elds 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 elds and functions of the classes it repre- sents.

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 eld list for the name of the member function, and nally 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 le 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 de ned 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 modi cation 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 modi ers and int is the base type. The type of x is represented by an SgDescriptType object that holds the information about the modi ers 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 de ned class. More speci cally, 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 isVarRefOf- Class(e, \myClass") that will return 1 when the class type matches and 0 otherwise. To write this function, we rst 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 nd 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 ow 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 ow analysis routines in Sage++ are still under development, and are subject to occasional modi cation. The current implementation is limited to Fortran 771.

## 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 infor- mation 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 ow analysis routines in Sage++ were not part of the initial Sage++ distribution.

## 4.2 Flow Analysis

Sage++ provides a framework for data ow analysis, to help users write their own ow analysis routines. For example, to implement an iterative forward data ow 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 o ers a more general ow analysis framework, with functions such as controlFlow(SgStatement \*stmt,...), which returns successors and predecessors of a statement in the control ow 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 ow analysis tools.

## 4.3 Loop Transformations

Sage++ provides a set of basic loop transformation tools. The following are some of the loop transfor- mations 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 di erent 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 speci ed 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](mailto:sage-request@cica.indiana.edu)

All the Sage++ papers, manuals, program les, 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](mailto: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 de ne 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 di erent parser generator that will, we hope, simplify this task.

# 7 Conclusions

The Sage++ object hierarchy provides a exible and extensible tool for manipulating an internal rep- resentation 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 e ciently, signi cantly 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 Speci cation." 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 scienti c 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 Ad- vanced Parallelizing Compiler for C and Fortran", In Languages and Compilers for Parallel Com- puting, 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), Cancu n, Mexico, Apr. 1994.

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

|  |  |  |
| --- | --- | --- |
| Top Level Block De nition Statements, F=Fortran(90), C=C(++) | | |
| Class Name | Parent Class | meaning |
| SgProgHedrStmt | SgProcHedrStmt | F: program stmt |
| SgProcHedrStmt | SgStatement | F: subroutine de nition |
| SgFuncHedrStmt | SgProcHedrStmt | C,F: function de nition |
| SgModuleStmt | SgStatement | F: module stmt |
| SgInterfaceStmt | SgStatement | F: module interface |
| SgBlockDataStmt | SgStatement | F: block data stmt |
| SgClassStmt | SgStatement | C: classfg |
| SgStructStmt | SgClassStmt | C: structfg |
| SgUnionStmt | SgClassStmt | C: unionfg |
| SgEnumStmt | SgClassStmt | C: enumfg |
| SgBasicBlockStmt | SgStatement | C: f g |
| Basic Control Statements | | |
| SgForStmt | SgStatement | F: do(), C: for()fg |
| SgWhileStmt | SgStatement | F: while(), C: while()fg |
| SgDoWhileStmt | SgWhileStmt | C: dofgwhile() |
| SgLogIfStmt | SgStatement | F: if()stmt |
| SgWhereStmt | SgLogIfStmt | F: where() stmt |
| SgIfStmt | SgStatement | F: if() then, C: if()fgelsefg |
| SgIfElseIfStmt | SgIfStmt | F: elseif() then |
| SgArithIfStmt | SgStatement | F: if() less,equal,greater |
| SgSwitchStmt | SgStatement | F: case, C:switch()fg |
| 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  SgValueExp SgKeywordValExp SgUnaryExp  SgCastExp SgDeleteExp SgNewExp SgExprIfExp SgFunctionCallExp SgFuncPntrExp SgExprListExp SgRefExp SgVarRefExp SgThisExp SgArrayRefExp SgPntrArrRefExp SgPointerDerefExp SgRecordRefExp SgStructConstExp SgConstExp SgVecConstExp SgInitListExp SgObjectListExp SgAttributeExp SgKeywordArgExp SgSubscriptExp SgUseOnlyExp SgUseRenameExp SgSpecPairExp SgIOAccessExp SgTypeExp SgSeqExp  SgStringLengthExp SgDefaultExp  SgLabelRefExp | The root class; also home for the basic binary operators:  +; ; ; =; %; <<; >>; <; >; <=; >=; &; j; &&; jj; + =; & =;  =; = =; % =; ^=, <<=; >>=; =; ==; ! =; >; ::; :  A base type (i.e. a \literal") value  Fortran keyword values in I/O statements, etc.  \*expr, & expr, sizeof(expr), expr, +expr, ++lhd, rhs++,  lhs, rhs , expr, and !expr (typename) expr;  delete [size] expr;  new typename (expr); (expr1) ? expr2 : expr3;  function name(expr1, expr2, );  (functionpointer)(expr1,expr2,expr3); Expression lists: expr1, expr2, ;  Fortran name references Scalar variable references C++ "this" reference  Fortran X(exp,exp) or C X[exp][exp];  Pointer used as array: (pointer)[index1][index2];  \*pointer; (also SgUnaryExp) StructName. eld;  Fortran 90 structure constructor; Fortran 90 array constructor;  [ exprlist ] ;  Used for initializations: f expr1,expr2, g;  Used for EQUIVALENCE, NAMELIST and COMMON statements Fortran 90 attributes  Fortran 90 keyword argument  vector range op: low:up:stride  Fortran 90 USE statement ONLY attribute Fortran 90 USE statement renaming  Fortran default control args to Input/Output stmts Fortran index variable bound instantiation  Fortran type expression  Fortran sequence expression Fortran string length expression Fortran default node  Fortran label reference |

[View publication stats](https://www.researchgate.net/publication/2725408)