A Principled, Complete, and Efficient Representation of C++

Gabriel Dos Reis[1]*, and Bjarne Stroustrup[2]

[1] Texas A&M University,
College Station, TX-77843, USA
gdr@cs.tamu.edu

[2] Texas A&M University,
College Station, TX-77843, USA
bs@cs.tamu.edu

Abstract

We present a systematic representation of C++, called IPR, for complete semantic analysis and semantics-based transformations. We describe the ideas and design principles that shaped the IPR. In particular, we describe how general type-based unification is key to minimal compact representation, fast type-safe traversal, and scalability. The IPR is general enough to handle real-world programs involving many translation units, archaic programming styles, and generic programming using likely C++0x extensions that affect the type system. The difficult issue of how to represent irregular (ad hoc) features in a systematic (non ad hoc) manner is among key contributions of this paper. The IPR can represent all of C++ with just slightly less than 200 node types; to compare the ISO C++ grammar has over 700 productions. Finally, we report impacts of this work on existing C++ compilers.

1 Introduction

The C++ programming language [11] is a general-purpose programming language, with bias toward system programming. It has, for the last two decades, been widely used in diverse application areas [20]. Beside traditional applications of general-purpose programming languages, it is being used in high-performance computing, embedded systems, safety-critical systems (such as, airplane controls), space explorations, etc. Consequently, the demand for static analysis and advanced semantics-based transformations of C++ programs is pressing. Dozens of analysis frameworks for C++ and for combination of C++ and code in other languages (typically C and Fortran) exist [1, 15, 16], but none handles the complete C++ language. Most are specialized to particular applications, and few (if any) can claim to both handle types and be portable across compilers.

This paper discusses a complete, efficient and direct representation of C++ implemented in C++, designed as part of a general analysis and transformation infrastructure, called *The Pivot*, being developed at Texas A&M University.

The IPR does not handle macros before their expansion in the preprocessor. With that caveat, we currently represent every C++ construct completely and directly. Note that by "completely" we mean that we capture all the type information, all the scope and overload information, and are able to reproduce input line-for-line. We capture templates (specializations and all) before instantiation — as is necessary to utilize the information represented by "concepts" [5, 10]. To be able to do this for real-world programs, we also

*Correspondence to: Postal Address(es) and Tel(fax) number(s).

handle implementation-specific extensions. We generate IPR from two compiler front ends [6, 9].

Our emphasis on completeness stems from a desire to provide a shared tool infrastructure. Complete representation of C++ is difficult, especially if one does not want to expose every irregular detail to every user. Some complexity is inherent, stemming from C++'s support of a wide range of programming styles; some is incidental, stemming from a long history of evolution under a wide range of real-world pressures; some originated in the earliest days of C. Independently of the sources of the complexity, a representation that aims to be general — aims to be a starting point for essentially every type of analysis and transformation — must cope with it. Each language feature — however obscure or advanced — not handled implies lack of support for some sub-community.

Our contribution is to engineer a small and efficient library with a regular and theoretically well-founded structure for completely representing a large irregular, real-world language. The IPR library has been developed side by side with a formalism to express the static semantics of C++.

2 Design Rules

The goals of generality directly guide the design criteria of IPR:

1. *Complete* — represents all Standard C++ constructs, but not macros before expansions, not other programming languages.
2. *General* — suitable for every kind of application, rather than targeted to a particular application area.
3. *Regular* — does not mimic C++ language irregularities; general rules are used, rather than long lists of special cases.
4. *Fully typed* — every IPR node has a type.
5. *Minimal* — its representation has no redundandent values and traversal involves no redundant dynamic indirections.
6. *Compiler neutral* — not tied to a particular compiler.
7. *Scalable* — able to handle hundreds of thousands of lines of code on common machines (such as our laptops).

Obviously, we wouldn't mind supporting languages other than C++, and a framework capable of handling systems composed out of parts written in (say) C++, C, Fortran, Java, and Python would be very useful to many. However, we do not have the resources to do that well, nor do we know if that can be done well. That is, we do not know whether it can be done without limiting the language features used in the various languages, limiting the kinds of analysis supported by the complete system, and without replicating essentially all representation node and analysis facilities for each language. These questions are beyond the scope of this paper. It should be easy to handle at least large subsets of dialects. In this context, C is a set of dialects. Most C++ implementations are de facto dialects.

Within IPR, C++ programs are represented as sets of graphs. For example, consider the declaration a function `copy`:

```
int∗ copy(const int∗ b, const int∗ e, int∗ out);
```

To represent this, we must create nodes for the various entities involved, such as types, identifiers, the function, and function parameters. parameters. Some information is implicit in the C++ syntax. For example, that declaration will occur in a scope, may overload other `copy` functions, and this `copy` may throw exceptions. The IPR makes all such information easily accessible to a user. For example, the IPR representation of `copy` contains the exception specification `throw(...)`, a link to the enclosing scope, and links to other entities called `copy` in that scope.

The types `int∗` and `const int∗` are both mentioned twice. The IPR library unifies nodes, so that a single node represents all `int`s in a program, and another node represents all `const int`s in a program, referring to the `int` node for its `int` part. The implication of this is that we can make claims of minimality of the size of the representation and of the number of nodes we have to traverse to gather information. It also implies that the IPR is not “just a dumb data structure”: it is a program that performs several fundamental services as it creates the representation of a program. Such services would otherwise have had to be done by each user or by other services. For example, the IPR implements simple and efficient automatic garbage collection.

The design of IPR is not derived from a compiler's internal data structures. In fact, a major aim of the IPR is to be compiler independent. The representation within compilers have evolved over years to serve diverse requirements, such as error detection and reporting, code generation, providing information for debuggers and browsers, etc. The IPR has only one aim: to allow simple traversals with access to all information as needed and in a uniform way. By *simple* traversal, we mean that the complexity of a traversal is proportional to the analysis or transform performed, rather than proportional to the complexity of the source language. Because the IPR includes full type information, full overload resolution, and full understanding of template specialization, it can be generated only by a full C++ compiler. That is, the popular techniques relying on just a parser (syntax analyser or slightly enhanced syntax analyser) are insufficiently general: They don't generate sufficient information for complete representation. The IPR is designed so as to require only minimal invasion into a compiler to extract the information it needs.

The IPR is a fully-typed abstract-syntax tree. This is not the optimal data structure for every kind of analysis and transformation. It is, however, a representation from which a specialized representation (*e.g.* a flow graph or an linkage specification) can be generated far more easily than through conventional parsing or major surgery to compiler internals. In particular, we are developing a high-level flow graph representation that can be held in memory together with the AST and share type, scope, and variable information.

3 Representation

Representing C++ completely is equivalent to formalizing its static semantics. Basically, there is a one-to-one correspondence between a semantic equation and an IPR node. The IPR does not just represent the syntax of C++ entities. The IPR essentially represents a superset of C++ that is far more regular than C++. Semantic notions such as overload-sets and scopes are fundamental parts of the library and types play a central role. In fact *every* IPR entity has a type, even types. Thus, in addition to supporting type-based analysis and transformation, the IPR supports concept-based analysis and transformation.

3.1 Nodes

Here, we do not attempt to present every IPR node. Instead, we present only as much of IPR as is needed to understand the key ideas and underlying principles. Each node represents a fundamental part of C++ so that each piece of C++ code can be represented by a minimal number of nodes (and not, for example, by a number of nodes determined by a parsing strategy).

3.2 Node design

The IPR library provides users with classes to cover all aspects of Standard C++. Those classes are designed as a set of hierarchies, and can be divided into two major groups:

1. abstract classes, providing interfaces to representations
2. concrete classes, providing implementations.

The interface classes support non-mutating operations only; these operations are presented as virtual functions. Currently, traversals use the Visitor Design Pattern [8] or an iterator approach.

IPR is designed to yield information in the minimum number of indirections. Consequently, every indirection in IPR is semantically significant. That is, an indirection refers to 0, 2 or more possibilities of different kinds of information, but not 1. For if there was only 1 kind of information, that kind of information would be accessed directly. Therefore an if-statement, a switch, or an indirect function call is needed for each indirection. We use virtual function calls to implement indirections. In performance, that is equivalent to a switch plus a function call [12]. Virtual functions are preferable for simplicity, code clarity, and maintenance.

The obvious design of such class hierarchies is an elaborate lattice relying on interfaces presented as abstract virtual base classes, and implementation class hierarchies, with nice symmetry between them. This was indeed our first design. However, that led to hard-to maintain code (prone to lookup errors and problems with older compilers), overly large objects (containing the internal links needed to implement virtual base classes), and slow (due to overuse of virtual functions).

The current design (described below) relies on composition of class hierarchies from templates, minimizing the number of indirections (and thus object size), and the number of virtual function calls. To minimize the number of objects and to avoid logically unnecessary indirections, we use member variables, rather than separate objects accessed through pointers, whenever possible.

Interfaces Type expressions and classic expressions can be seen as the result of unary, or binary, or ternary node constructors. So, given suitable arguments, we need just three templates to generate every IPR node for "pure C++". In addition, we occasionally need a fourth argument to handle linkage to non-C++ code, requiring a quaternary node. For example, every binary node can be generated from this template:

```
template<class Cat = Expr,  // kind (category) of node
         class First = const Expr&,
         class Second = const Expr&>
struct Binary : Cat {
   typedef Cat Category;
```

```
    typedef First Arg1_type;
    typedef Second Arg2_type;
    virtual Arg1_type first() const = 0;
    virtual Arg2_type second() const = 0;
};
```

`Binary` is the base class for all nodes constructed with two arguments, such as an array type (node) or an addition expression (node). The first template parameter `Cat` specifies the kind (category) of the node: (classic) expression, type, statement, or declaration. The other two template parameters specify the type of arguments expected by the node constructor. Most node constructors take expression arguments, so we provide the default value `Expr`. The functions `first()` and `second()` provide generic access to data.

Note how `Binary` is derived from its first argument (`Cat`). That's how `Binary` gets its set of operations and its data members: It inherits them from its argument. This technique is called "the curiously recurring template pattern" [4] or "the Barton-Nackman trick"; it has been common for avoiding tedious repetition and unpleasant loopholes in type systems for more than a decade (it is mentioned in the ARM [7], but rarely fails to surprise). The strategy is systematically applied in the IPR library, leading to linearization of the class hierarchy (see Figure 1).

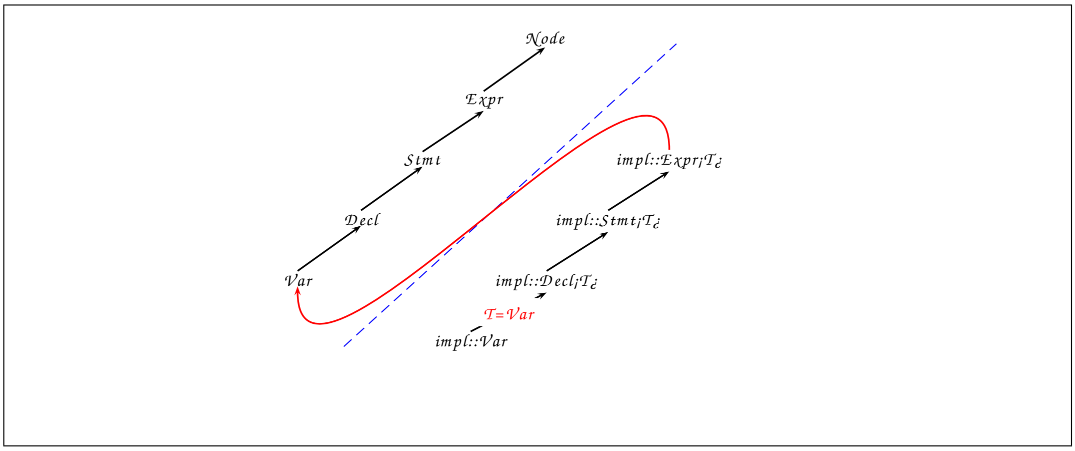

Figure 1: Current design of the IPR library

A specific interface class is then derived from the appropriate base (`Unary`, `Binary`, or `Tertiary`). For example:

```
struct Array
      : Binary<Category<array_cat,Type>, const Type&> {
    Arg1_type element_type() const { return first(); }
    Arg2_type bound() const        { return second(); }
};
```

That is, an `Array` is a `Type` taking two arguments (a `Type` and an `Expr`) and a return type (a `Type`). `Array`'s two member functions provide the obvious interface: `element_type()` returns the type of an element and `bound()` returns the number of elements. Please note that the functions `element_type()` and `bound()` are themselves *not* virtual functions; they are simple "forwarding" inline functions, therefore induce no overhead.

The category argument `Category<array_cat,Type>` exposes an implementation detail. The category is `Type` (i.e., an array is a type), but to optimize comparisons of types, we associate an integer `array_cat` with the `Array` type. Logically, it would be better not to expose this implementation detail, but avoiding that would involve either a per-node memory overhead storing the `array_cat` value or a double dispatch in every node comparison. We introduced `array_cat` after finding node comparison to be our main performance bottleneck. So far, we have found no systematic technique for hiding `array_cat` that doesn't compromise our aim to keep the IPR minimal.

Concrete Representations Each interface class has a matching implementation class. Like the interface classes, the (concrete) implementation classes are generated from templates. In particular, `impl::Binary` is the concrete implementation corresponding to the interface `ipr::Binary`:

```
template<class Interface>
struct impl::Binary : Interface {
   typedef typename Interface::Arg1_type Arg1_type;
   typedef typename Interface::Arg2_type Arg2_type;
   struct Rep {
      Arg1_type first;
      Arg2_type second;
      Rep(Arg1_type f, Arg2_type s)
         : first(f), second(s) { }
   };
   Rep rep;

   Binary(const Rep& r) : rep(r) { }
   Binary(Arg1_type f, Arg2_type s) : rep(f, s) { }

   // Override ipr::Binary<>::first.
   Arg1_type first() const { return rep.first; }

   // Override ipr::Binary<>::second.
   Arg2_type second() const { return rep.second; }
};
```

The `impl::Binary` implementation template specifies a representation, constructors, and access functions (`first()` and `second()`) for the `Interface`. Given `impl::Binary`, we simply define `Array` as a `typedef` for the implementation type:

```
typedef impl::Binary<impl::Type<ipr::Array> > Array;
```

The `Array` type is generated as an instantiation of the `Binary` template.

3.3 Sharing

By *node sharing*, we mean that two nodes that represent the same entity shall have the same address. In particular, node sharing implies that if a node constructor is presented twice with equal lists of arguments, it will yield the same node. If node sharing is implemented for a class, that class is said to be *unified.* Since a user-defined type (classes or enums) can be defined only once in a given translation unit, sharing of nodes is suggested by C++ language rules. Every IPR node can be unified; exactly which are unified is a design choice related to performance (of the IPR itself and of applications). This can be used to tune IPR.

Implementing node sharing is easy for named types, but less straightforward for built-in types and types constructed out of other types using composition operators (e.g., `int`, `double (*)(double)`, and `vector<Shape*>`). The problem arise because such types are not introduced by declarations. They can be referred to without being explicitly introduced into a program. For example, we can say `int*` or take the address of a `double` and implicitly introduce `double*` into our set of types. Node sharing for such types implies maintenance of tables that translate arguments to constructed nodes. Since an expression does not have a name, unifying expression nodes share this problem (and its solution) with nodes for unnamed types.

We can define node sharing based on at least two distinct criteria: syntactic equivalence, or semantic equivalence. Node sharing based on syntactic equivalence has implications on the meaning of overloaded declarations; two declarations might appear overloaded even though only the spelling of their types differs. For example, the following function template declarations are possibly overloads whereas Standard C++ rules state they declare the same function.

```
template<typename T, typename U>
  void bhar(T, U);

template<typename U, typename T>
  void bhar(U, T);
```

The reason is that for templates, only the positions of template-parameters (and their kinds) are relevant. Normally, we do not care whether the name of a template-parameter is `T` or `U`; however, in real programs, people often use meaningful names, such as `ForwardIterator` instead of `T`.

3.4 Effects of unification

Space is time. It should be obvious that, because nodes are not repeatedly created to represent the same type, node sharing leads to reduced memory usage and less scattering in the address space (and therefore few cache misses.) Experiments with the classic first program

```
#include <iostream>
int main()
{
   std::cout << "Hello, World" << std::endl;
}
```

based on GCC-3.4.2 — at the time we started the IPR project — reveal that, in non-sharing mode, there are 60855 calls to type constructors; out of which we have

1. 60% for named types (only less than 1% are syntactically distinct),
2. 17% for pointer types,
3. 11% for `const`-qualified types,

Due to curiosities in the GCC compiler infrastructure, we cannot get precise counting of nodes, so the above are approximates (±5%). However, the GCC representation was about 32 times the size of the IPR representation. The "Hello, World!" program is useful because it drags in so much relatively advanced code though its `#include`. However, even for medium-sized programs we must multiply the figures by at least 100 to get realistic measures, and then our savings in time and space begin to appear significant. Once we start to represent multiple translation units simultaneously, unification becomes a critical component of scalability.

Inspired by our design and our measurements, GCC has switched to a unified internal representation of types.

For program analysis that requires type comparison, node sharing offers time efficiency because type comparison is reduced to pointer comparison. This is significant because many forms of analysis (as well as the IPR itself) basically boil down to "traverse the program representation doing a lot of comparisons along the way to decide which nodes need attention". With node sharing, those comparisons are simple pointer comparisons. Without node sharing they are recursive double dispatch operations. Obviously, the time gained sharing nodes should be weighted against the overhead of building and using hash tables to be able to find existing nodes when you make a new node.

In the context of program transformation, another advantage of node sharing is consistency. Since there can be only one node for a type named `Foo`, we never need to walk through the whole graph to modify the properties of all `Foo`s. That is an important property when merging separately compiled translation units, doing whole-program analysis, and doing systematic substitutions. For example, with a single substitution, we can replace all uses of a type, say `int[]`, with another type, say `vector<int>`, in a whole translation unit.

4 Details

"The devil is in the details." If C++ had been designed yesterday with "simple complete representation" as a major goal, representing it would have relatively been easy. Basically, the previous section would have been the end of the story. However, elements of C++ were designed more than 30 years ago (for pre-K&R C) and much (both standard and non-standard) have been added since. This seriously complicates the design of a complete representation for C++. However, these "details" must be dealt with to produce a tool, rather than a toy. If you feel like commenting "C++ is just too complicated, let's work on tools for another language" then think what our favorite language might look like after 20 years of serious industrial use and also consider the number of people that will benefit from extra work required for a mature language.

Dealing with "details" has been much more than 50% of the total design effort. The "details" are plentiful and irregular. However, we must fit them into a more general framework so that the IPR user do not need to remember (and handle) a long list of special cases. In other words, we cannot take an ad hoc approach to dealing with ad hoc language features. We must abstract the many "details" into a few IPR constructs.

4.1 Lexical and home scopes

A name can simultaneously belong to more than one scope. For example:

```
int f(int i) {
   extern int g(int);     // g is global
   return g(i);
}
```

```
class A {
   friend void f(A) { }   // f is in enclosing scope; visible
                          // only through ‘‘argument dependent’’
                          // name lookup
};

namespace N {
   extern "C" void bar();  // bar is global
}
```

In the function `f(int)`, the locally declared function `g(int)` is visible only in the local block established by the body of `f(int)`. However, it really belongs to the global scope; that is, there is no nested or local functions in C or C++. The function `f(A)` defined in the class `A` really belongs to the enclosing namespace scope of `A`. However, an ordinary name lookup will not find it (unless a matching declaration is also available in that scope, which is not the case here). That function is visible only through a special name lookup (*argument dependent name lookup*) that considers the syntactic form of a call and the type of the arguments. The third example declares the function `bar()` as having a "C" language calling convention, consequently it really belongs to the global scope. However name lookup will not find it in the global scope – it is visible only the scope of `N`. Note also that there can be only one such function in the whole program named `bar` with that same type and "C" calling convention.

Note that the first example is also C and fairly common in C-style C++ code.

The general solution to all of these problems (and more) is that every declaration has a *lexical scope* and some also have a *home scope.* All information relating to the entity declared can be found though its entry in its home scope.

```
struct Decl : Stmt {
   // ...
   virtual const Name& name() const = 0;
   virtual const Region& home_region() const = 0;
   virtual const Region& lexical_region() const = 0;
   // ...
};
```

The lexical region is the scope in which the declaration appear in the source text. The home region the scope in which the declaration really belongs to according to the C++ rules. For most cases, those two regions are the same. However, for each of the examples above the home region and lexical region differ.

4.2 Overloading, specialization, etc.

Often, several declarations are related. For example, a function can have several declarations (which must match) and several functions in a scope can have the same name (so that they must be considered together for overload resolution). Of course, IPR must keep the information that the programmer provided (the many declarations), but it must also present a single entity (the function, the variable, the template) to the user unless the user express an interest in "the details". Consider:

```
void print(double);

void f(int i) { print(i); }
void print(int);

void g(int i) { print(i); }

void print(double d) { cout << d; }
```

The IPR represents different functions with the same name in the same scope as overload sets, but different declarations of the same function are linked to the first declaration of that function: The `Decl` class handles all linked declarations with just three functions:

```
struct Decl : Stmt {
   // ...
   virtual const Decl& master() const = 0;
   virtual const Sequence<Decl>& decl_set() const = 0;
   virtual const Decl& defining_decl() const = 0;
   // ...
};
```

The `master()` is the first declaration of a given name encountered. The `decl_set()` is the set of all declarations of that name. The `defining_decl()` is the defining declaration.

Both the primary declaration and all the secondary declarations are placed in their proper scopes and their proper places in that scope. This is essential: Note how you can change the meaning of the program fragment above by reordering the declarations. This is unfortunate, but follows directly from the C++ standard and is used in real code.

The distinction between an overload set and a linked set of declarations of the same entity also directly reflects the C++ distinction between overloading and specialization.

4.3 Lowering

Even at the level of an AST, different users want different levels of representation. We have already "lowered" the representation of the program by expanding macros, so that the IPR represents a compiler's view of a program, rather than the view of a programmer looking at a screen. This is an important design decision for IPR and not one that's always ideal. However, we don't think we had much choice. Macros are inherently irregular, so that distinctions among fundamental notions — such as, declaration, statement, and expression — are often blurred by macros.

The next major design choice is whether to retain `typedef` names. For example:

```
typedef int Length;
Length x;
```

Is `x` a `Length` or an `int`. According to C++, it's an `int` because a typedef name is only an alias. However, `Length` has a meaning to some programmer and some forms of analysis assign meaning to typedef names (and to other aliases). "Other aliases" include namespace aliases, using declarations, and (in C++0x) template aliases. It is important that the IPR implement a uniform policy vis a vis aliases.

Finally, there is the issue of how to represent member access and uses of overloaded operators. Consider:

```
void f(T x, TT p) {
     ++x;
     T(x) = 5;
     p->f();
}
```

We could represent ++x as a use of operator ++ or as call node for the function `operator++()`. The first alternative is the user's view, the syntactic view. The latter view is "lowered" to reflect a semantic view. For example, lowering to a uniform function call notation simplifies programs concerned with program execution.

It is important to have a uniform policy on this kind of examples. Several times we (as have others) thought we had a free choice in such decisions for a specific operator, language construct, or type. In fact, we do not. Consider the case where the example above is a template function with `T` as an unconstrained template parameter. In that case, we cannot even know whether `T(x)` is a cast or a declaration of a variable `x` with redundant parentheses! Any uniform policy in a system that fully handles templates must retain the syntactic view – any lowering will be premature. Also, the syntactic view is the only one that allows re-generation of the user's code without risk of subtle semantic changes. For example, if we transformed ++x to a uniform call syntax (say) `operator++(&x)`, we would not (without additional information) know whether the user wrote ++x or `x.operator++()` or `operator++(&x)`.

So, to preserve information and thereby support a larger set of applications, the IPR doesn't lower by default. If you want lowering, you can ask IPR to do so at creation time. This works well, but even that may be a premature optimization and we are considering replacing the option to lower by a lowering IPR-to-IPR tool.

Note that before lowering, IPR will take a purely syntactic view of aliases. For example:

```
typedef int Length;
// ...
void f(Length);
void f(int);
```

Before lowering, the IPR – like a naive human reader – will think that there are two functions (syntactic equivalence) whereas after lowering it will realize that there really (according to C++) is only one. The distinction can be useful for some forms of analysis.

4.4 Proprietary extensions

Most compiler providers have a host of proprietary language extensions that the average end user doesn't see. However, the deep internals of most standard libraries are littered with them. Try representing the innocent-looking "Hello, world!" program:

```
#include<iostream>

int main()
{
     std::cout << "Hello, world!";
}
```

To do this, we have to handle dozens of proprietary extensions. Such extensions (of course) differ from provider to provider and it is not unusual that they vary from release to release. They tend to be plentiful in the lowest levels of code (OS interfaces, I/O, memory management, etc.), so the standard headers included to compile "Hello, world!" is a good place to look for them. For example, in <iostream> from GCC-4.3.0, we find five extensions in what should have been a simple one-line function declaration:

```
extern int
snprintf (char *__restrict __s, size_t __maxlen,
            __const char *__restrict __format, ...)
    throw () __attribute__ ((__format__ (__printf__, 3, 4)));
```

Often, such extensions are hidden from the programmers by wrapping them in macros, but the IPR sees through that. To deal with this, we have temporarily been reduced to the "ad hockery" of simply adding IPR nodes to represent the proprietary extensions, usually one new node per extension. Given the rate of change in these extensions, this approach is not sustainable. The "Hello, world!" program is portable and by default the IPR for it should also be. The solution is to modularize the program so that we don't represent "details" of `<iostream>` in the IPR unless the user explicitly requests it. Such requests may be non-portable in the sense that an IPR implementation for a given compiler (or version of a compiler) may not be configured to precisely handle all proprietary extensions.

Please note that not handling proprietary extensions is not an option for a general representation, such as IPR, even though it can be for a specialized representation (say) aimed at the specific task of parallelizing array computations.

4.5 Separate compilation and whole-program analysis

Real-world C++ programs consist of many separately-compiled translation units. Each translation unit often consists of many hundreds of header files recursively `#include`d by a single source code file. As described so far, the IPR represents a C++ translation unit as it appears after preprocessing; that is, as a single source file with the information from the header files included and macros expanded. We can handle multiple translation units by storing the IPR for many units and then reading them back in. The ability to store the IPR in what we call XPR ("eXternal Program Representation") format is essential because most C++ compilers cannot compile two translation units in a single invocation.

The fact that IPR is unified is most useful here because that way every inconsistency between translation units is automatically caught. For example, we could try to generate IPR for a program with the two source files

```
// x.cpp
int glob;
int gfct();
```

and

```
// y.cpp
double glob;
void gfct();
```

The IPR will detect the two errors.

So, the IPR (supported by XPR) trivially supports whole-program analysis: Just add as many source files as you want and run traversals and transforms as usual. This is also the point where the compactness of nodes and the space savings from unification really pays off.

However, the situation is still not quite ideal. Considering the problems with proprietary extensions deep in implementations, we must consider an explicit approach to modularity. The IPR knows the source of every declaration (to the line number), so it is easy to tell what interface to a "module", such as `<iostream>` was really used by user code. This implies that we could represent a use of a module as the name of its header file plus the set of declaration nodes used to access it. That is, we can treat a header file as a parameterized module. Generating that is a fairly simple IPR program, but the need to abstract from details of header files is so common that we are considering integrating it into the IPR itself.

Note that in general two uses of a “module” represented as a header file are not equivalent because macros, typedef, etc. can affect the set of definitions in the header and meaning of those definitions. We can trivially use the IPR to detect any differences or to detect any differences that matter for a given use, though.

4.6 Simplicity

One measure of simplicity is that the complete source code for IPR (excluding compiler-to-IPR generators) is just 2,500 lines of C++ (excluding comments). The code for IPR is available from the author.

5 Related and Future Work

The IPR was inspired by the *eXtented Type Information* library designed by the second author. XTI focused on the representation of the C++ type system, whereas IPR aims at the full C++ language. There are *many* projects [1, 2, 14, 15, 17, 18, 19] targeting static analysis and transformations of C++ programs. For example, CodeBoost [2, 3, 13] focuses on transformations of C++ programs, for numerical PDE solvers, written in the Sophus style. Simplicissimus [18, 19] and ROSE [17] are other projects for transforming C++ programs. Many of these systems are commercial and not documented in the literature. Few aim to handle full Standard C++, few aims at generality (as opposed to specific applications), and few aims at compiler independence. None — to our knowledge — aims at all three.

Obviously, our immediate aims include applications that test the generality and portability of the IPR and its associated tools. For example, conventional style analyzers, statistics gathering, and visualization tools. We have been able to represent the full source code FireFox. We will experiment with the use of concepts and library-specific validations, optimizations, and transformations in the domains of parallel, distributed, and embedded systems.

We plan to provide more ways of specifying traversals and transforms (such as ROSE and CodeBoost) and to work on better ways of specifying type-sensitive (incl. concept sensitive) traversals and transformations.

From the standpoint of the structure of the IPR, the most important direction of work is to get a better handle on modularity.

We will work to make the compiler to IPR generation more complete; it is already more complete that some popular compilers, but every lacking feature will cause a problem for someone. In addition we will try to interface the IPR to more compilers and handle more dialects.

6 Conclusion

Current frameworks for representing C++ are not general, complete, accessible and efficient. In this paper, we have shown how general, systematic, and simple design rules can lead to a complete, direct, and efficient representation of ISO Standard C++. In particular, we don’t have to resort to ad hoc rules for program representation or low-level techniques for completeness or efficiency. Unification helps maintain consistency, keeps our program representation compact (as required for scalability), and minimizes the cost of comparisons. To serve the widest range of applications, we use syntactic unification. Given syntactic unification, we can implement semantic unification by a simple transformation, whereas the other way around is impossible without referring back to the program source text. In addition to unification, careful and systematic node class and node class hierarchy design is necessary to minimize overhead and enable scaling.

Acknowledgements

This work was partly supported by NSF grant CCF-0702765.

References and Notes

[1] S. Amarasinghe, J. Anderson, M. Lam, and C.-W. Tseng. An overview of the SUIF compiler for scalable parallel machines. In *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, San Francisco, CA, 1995.

[2] O. Bagge. CodeBoost: A Framework for Transforming C++ Programs. Master's thesis, University of Bergen, P.O.Box 7800, N-5020 Bergen, Norway, March 2003.

[3] O. Bagge, K. Kalleberg, M. Haveraaen, and E. Visser. Design of the CodeBoost transformation system for domain-specific optimisation of C++ programs. In Dave Binkley and Paolo Tonella, editors, *Third International Workshop on Source Code Analysis and Manipulation (SCAM 2003)*, pages 65–75, Amsterdam, The Netherlands, September 2003. IEEE Computer Society Press.

[4] James O. Coplien. Curiously Recurring Template Patterns. *C++ Report*, 7(2):24–27, 1995.

[5] Gabriel Dos Reis and Bjarne Stroustrup. Specifying C++ Concepts. In *Conference Record of POPL '06: The 33th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 295–308, Charleston, South Carolina, USA, 2006.

[6] The Edison Design Group. `http://www.edg.com/`.

[7] Margaret E. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, 1990.

[8] Erich Gamma, Richard Helm, Ralph Johson, and John Vlissides. *Design Patterns.* Addison-Wesley, 1994.

[9] GNU Compiler Collection. `http://gcc.gnu.org/`.

[10] Douglas Gregor, Jaakko Järvi, Jeremy Siek, Bjarne Stroustrup, Gabriel Dos Reis, and Andrew Lumsdaine. Concepts: Linguistic Support for Generic Programming in C++. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programming Languages, Systems, and Applications*, pages 291–310, New York, NY, USA, 2006. ACM Press.

[11] International Organization for Standards. *International Standard ISO/IEC 14882. Programming Languages — C++*, 2nd edition, 2003.

[12] International Organization for Standards. *ISO/IEC PDTR 18015. Technical Report on C++ Performance*, 2003. Performance.

[13] K. Kalleberg. User-configurable, High-Level Transformations with CodeBoost. Master's thesis, University of Bergen, P.O.Box 7800, N-5020 Bergen, Norway, March 2003.

[14] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO '04: Proceedings of the international symposium on Code generation and optimization*, page 75, Washington, DC, USA, 2004. IEEE Computer Society.

[15] Sang-Ik Lee, Troy A. Johnson, and Rudolf Eigenmann. Cetus — An Extensible Compiler Infrastructure for Source-to-Source Transformation. In *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, pages 539–553, October 2003.

[16] Georges C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. CIL: Intermediate Language and Tools for Analysis and Tranformations of C Programs. In *Proceedings of the 11th International Conference on Compiler Construction*, volume 2304 of *Lecture Notes in Computer Science*, pages 219–228. Springer-Verlag, 2002. `http://manju.cs.berkeley.edu/cil/`.

[17] M. Schordan and D. Quinlan. A Source-to-Source Architecture for User-Defined Optimizations. In *Proceeding of Joint Modular Languages Conference (JMLC'03)*, volume 2789 of *Lecture Notes in Computer Science*, pages 214–223. Springer-Verlag, 2003.

[18] S. Schupp, D. Gregor, D. Musser, and S.-M. Liu. User-extensible simplification — type-based optimizer generators. In R. Wihlem, editor, *International Conference on Compiler Construction*, Lecture Notes in Computer Science, 2001.

[19] S. Schupp, D. Gregor, D. Musser, and S.-M. Liu. Semantic and behavioural library transformations. *Information and Software Technology*, 44(13):797–810, October 2002.

[20] Bjarne Stroustrup. C++ Applications. `http://www.research.att.com/~bs/applications.html`.