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

Gabriel Dos Reis and Bjarne Stroustrup

Abstract. We present a systematic representation of C++, called IPR, for complete semantic analysis and semantics-based program 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. For example, the representation of a fairly typical non-trivial C++ program in GCC 3.4.2 was 32 times larger than its IPR representation; this led to significant improvements to GCC. IPR is general enough to handle real-world programs involving many translation units, archaic programming styles, and generic programming using 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 the key contributions of this paper. The IPR data structure can represent all of C++ with just 157 simple node types; to compare the ISO C++ grammar has over 700 productions. The IPR is used for a variety of program analysis and transformation tasks, such as visualization, loop simplification, and concept extraction. Finally, we report impacts of this work on existing C++ compilers.

1. Introduction

The C++ programming language [16] 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 [32, 33, 31]. Besides traditional applications of general-purpose programming languages, it is being used in high-performance computing, embedded systems (such as cell phones and wind turbines), safety-critical systems (such as airplane controls), space exploration, etc. Consequently, the demand for static analysis and advanced semantics-based transformations of C++ programs is pressing. That in turn calls for scalable infrastructures capable of

representing and processing large real world programs. Dozens of analysis frameworks for C++ programs, and for programs written in a combination of C++ and other programming languages (typically C and Fortran) exist [24, 1, 22], but none handle the complete C++ language. Most analysis frameworks are specialized to particular applications — e.g. "class browsing", a particular C++ front-end representation — and few (if any) can claim to both handle types and be portable across compilers.

A scalable infrastructure for analyzing large programs written in a language as complex as C++ must be well engineered but also requires more than "just engineering" and "implementation tricks." This paper discusses a principled, complete, and efficient data structure for direct representation of C++ programs. It is implemented in C++, and designed as part of a general analysis and transformation infrastructure, called *The Pivot*, developed at Texas A&M University and used for research there and in a few other places. In particular, *The Pivot* aims at supporting high-level parallel and distributed programming techniques. It consists of:

1. data structures for Internal Program Representation (IPR);
2. a persistent form named eXternal Program Representation (XPR);
3. tools for converting between IPR and XPR;
4. general traversal and transformation tools.

In addition, there are IPR generator compiler interfaces. Those serve as the building blocks for specific tools such as IDL generators, style checkers, etc. An in-depth coverage of the *The Pivot* [7] infrastructure is postponed to future publication. Rather, the main focus of this paper is the design principles and implementation of the central data structure of *The Pivot*.

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 they are instantiated — as is necessary to utilize the information represented by "concepts" [9, 15]. To be able to do this for real-world programs, we also handle implementation-specific extensions. We currently generate IPR from the GCC [14], EDG [10], and Clang [5] front ends.

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 define, implement, and refine 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++[8].

2. Design Rules

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

1. *Complete* — represents all ISO 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 redundant values and traversal involves no redundant dynamic indirections.
6. *Compiler neutral* — not tied to any particular compiler.
7. *Scalable* — able to handle hundreds of thousands of lines of code on common machines (such as our laptops).

Obviously, we would not mind supporting languages other than C++, and a framework capable of handling systems composed out of parts written in (say) C++, C, Fortran, Java, C#, 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 nodes 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, the C programming language is a set of dialects. Most C++ implementations are de facto dialects [4].

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

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

which presumably copies elements in the sequence [`b`,`e`) into the sequence whose start is designated by `out`. To represent that function declaration, we must create nodes for the various entities involved, such as types, identifiers, function parameters, etc. Some information is implicit in the C++ syntax. For example, this 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 instance, the IPR representation of `copy` contains the exception

specification `throw(...)` — meaning can throw an exception of any type — a link to the enclosing scope, and links to other entities (e.g. overloads) called `copy` in that scope.

The types `const int*` and `int*` are both mentioned twice: `const int*` for the first two parameters, and `int*` for the third parameter and the return type. To reduce redundancy, 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 library 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 libraries. For instance, the IPR implements a simple and efficient automatic garbage collection.

The design of IPR is not derived from any compiler's internal data structures. In fact, a major aim of the IPR is to be compiler independent. Representations 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 (for example, see §6.7). 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 analyzer or slightly enhanced syntax analyzer) are not adequate: They do not 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 more specialized representations (*e.g.* a data flow or control flow graph) 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. The IPR language

The C++ programming language as defined by the ISO standard [16] is complex. A viable representation must resist the temptation of exposing all irregularities. A complete representation cannot afford to provide support only for a "nice" subset. Consequently, the IPR seeks to present a regular *superset* of ISO C++,

with faithful semantics. The language defined by IPR nodes consists mostly of expressions. IPR expressions are generalizations of C++ expressions. We will refer to the latter as classic expressions. IPR expressions are divided into four major categories: classic expressions, types, statements and declarations. Here, we present only the representation of the type system. The semantics can be found in [9] and [8].

Using τ for types, ϵ for expressions, δ for declarations, and $\vec{\bullet}$ for sequence of $\bullet$, the C++ type system is modeled as a multi-sorted algebra shown in Figure 1.

	IPR nodes	Example
$\tau ::=$	*Pointer*(τ)	`T*`
—	*Reference*(τ)	`T&`
—	*Array*(τ, ϵ)	`T[68]`
—	*Qualified*(cv, τ)	`const T`
—	*Function* (τ, τ, τ)	`int (int, int) throw()`
—	*Class*($\vec{\delta}$, $\vec{\delta}$)	`class B : A { int v; }`
—	*Union*($\vec{\delta}$)	`union { int i; double d; }`
—	*Enum*($\vec{\delta}$)	`enum { bufsize = 1024 };`
—	*Namespace* ($\vec{\delta}$)	`namespace { int count; }`
—	*Decltype*(ϵ)	`decltype(count)`
—	*As_type*(ϵ)	`int`
—	*Template* (τ, τ)	*class template*
—	*Product*($\vec{\tau}$)	*function parameter-type list*
—	*Sum*($\vec{\tau}$)	*exception specification list*
$cv ::=$	None	*no cv-qualifier*
—	Const	`const`
—	Volatile	`volatile`
—	Restrict	`restrict` // *not ISO C++*

Figure 1: Abstract syntax of IPR nodes for the C++ type system

The type constructors *Pointer*, *Reference* and *Array* correspond to the usual operations for constructing pointers, references, and array types. The operations *points_to*, *refers_to*, *element_type* and *bound* extract the arguments used to construct such types according to the equations

$$\begin{aligned} \mathit{points\_to}\,(\mathit{Pointer}\,(\tau)) &= \tau\ , \\ \mathit{refers\_to}\,(\mathit{Reference}\,(\tau)) &= \tau\ , \\ \mathit{element\_type}\,(\mathit{Array}\,(\tau, \epsilon)) &= \tau\ , \\ \mathit{bound}\,(\mathit{Array}\,(\tau, \epsilon)) &= \epsilon\ . \end{aligned}$$

The *Decltype* constructor gives the "declared" type of an expression. The type constructor *As_type* turns an arbitrary expression into a type. A type constructed

by *Decltype* or *As_type* supports the operation *expr* which yields the argument used to construct that type:

$$expr\,(Decltype\,(\epsilon)) = \epsilon,\; expr\,(As\_type\,(\epsilon)) = \epsilon.$$

The `decltype` operator is part of C++0x's support for generic programming [18]. We use *As_type* to introduce built-in types and type variables within IPR, and to handle dependent types in template declarations.

Product and *Sum* types do not explicitly exist in C++. However they are notions informally used by C++ programs, which are useful for a formal specification. For example, we use *Product* to represent lists of function parameter types. The *Sum* type constructor is dual to *Product* [23]. It represents a collection of types supporting a set of common operations. For example, we use *Sum* to represent union members and members of exception specifications. Both *Product* and *Sum* supports the subscription and *size* operations. The operation *size* reports the number of types in the product or sum.

$$\begin{aligned} size\,(Product\,(s)) &= size\,(s)\ , \\ size\,(Sum\,(s)) &= size\,(s)\,, \\ Product\,(s)_{\mathtt{i}} &= s_{\mathtt{i}}\ , \\ Sum\,(s)_{\mathtt{i}} &= s_{\mathtt{i}}\ . \end{aligned}$$

Unlike ISO C++, the IPR considers that a template declaration has a a type. The *Template* constructor takes as arguments a parameter-type list in form of a product type, and a type to be parameterized. Note that this generalization allows us to parameterize any declaration, including variable and namespace declarations. The IPR aims for greater generality and regularity than what C++ currently offers.

Given a template, we can retrieve its parameters using *parameters* and the parameterized type using *parameterized*:

$$\begin{aligned} parameters\,(Template\,(p,\tau)) &= p, \\ parameterized\,(Template\,(p,\tau)) &= \tau. \end{aligned}$$

For example, given (the node representing) the class template

```
template<typename T, int N>
  struct Buffer {
     T data[N];
  };
```

parameterized will return (the node representing) the class-expression:

```
struct { T data[N]; }
```

We consider a uniform, complete, and universally applicable representation of a programming language the ultimate (and in general unobtainable) ideal for compiler and tools developers. We urge the reader to resist the temptation of concluding, from the presentation given so far, that a uniform and complete representation of C++ is easy. It is not. Several obstacles have to be overcome, and irregularities must be embedded in more general structures, hiding complexity from users, yet

retaining the standard semantics of C++ (e.g. see [4]). Some of these challenges are discussed in greater detail in §6.

4. The Role of Algebra and Analysis

A large number of C++ expressions share the same structure. For example, `x + y` is a binary expression whose representation is similar to that of `dynamic_cast<const T*>(p)`, and to that of the type expression `int [32]`. They are all binary operators. In the design and implementation of the IPR, we adopt a systematic algebraic view that captures those similarities. This algebraic view naturally leads to structures that are parameterized by the type of their components. Templates in C++ are the primary abstraction tools to deal with "algebra".

For a particular usage, the user may not necessarily be interested in the exact algebraic structure of a particular IPR node. Rather, she might be more interested in whether the node represents an expression, or a declaration, or a member function definition. So, we need an approximation mechanism to cut down on the amount of (sometimes overwhelming) information that the algebraic view gives us. We need a mechanism for selective ignorance of details. Which is what Analysis is really all about. Class inheritance in C++ is the primary mechanism for "approximation". Base classes provide an initial estimate that get refined by derived classes. A carefully engineered combination of templates ("Algebra") and class inheritance ("Analysis") is at the core of the IPR implementation as we explain in the next section. The end goal being a principled, complete, efficient representation of the *semantics* of a C++ program, *i.e.* the geometry of a program, which is usually expressed as a linear sequence of characters (common concrete syntax.)

5. 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 primarily represent the syntax of C++ entities. It 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.

5.1. 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. The IPR is a direct representation of the C++ semantics rather than a direct representation of its syntax, so calling it an AST is a bit of a misnomer (unless

you – unconventionally – think of the 'S' as standing for "semantic"). 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).

5.2. Node design

The IPR library provides users with classes to cover all aspects of ISO 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 [13] or an iterator approach [35].

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 or faster than a switch plus a function call [17]. Virtual functions are preferable for simplicity, code clarity, and maintenance.

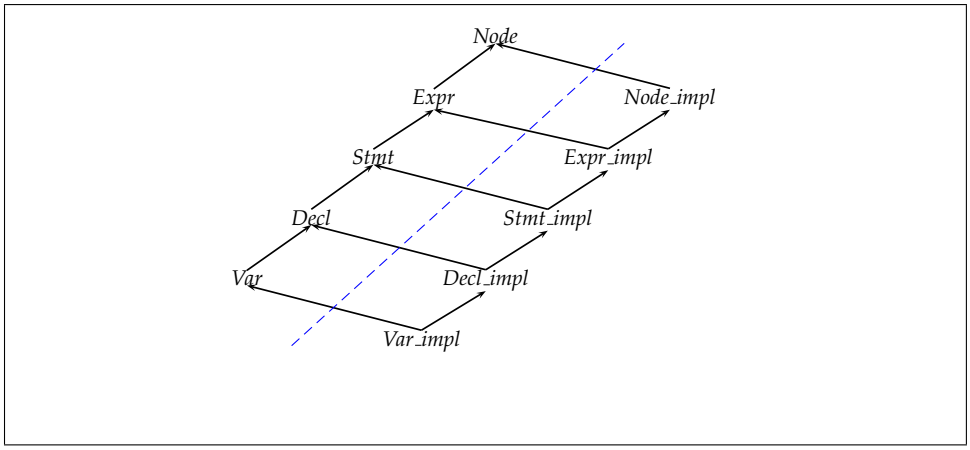

Figure 2: Early design of the IPR class hierarchy

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 — see Figure 2. 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). These overheads are indicators that the "obvious" design fails to meet IPR's fundamental design criterion to have each indirection be semantically significant: Some of the internal structure used to support virtual function calls unnecessarily delay choices among alternatives until run time.

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. This implementation strategy reflects a combined algebraic and analytic view (see §4) of a program representation. 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.

5.2.1. 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 (or 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 is 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" [6] or "the Barton-Nackman trick"[1]; it has been common for avoiding tedious repetition and unpleasant loopholes in type systems for two decades (it is mentioned in the

[1] variations of this technique are usually referred as Barton-Nackman trick; the essence was documented in [11]

ARM [11], but rarely fails to surprise). The strategy is systematically applied in the IPR library, leading to linearization of the class hierarchy (see Figure 3). A

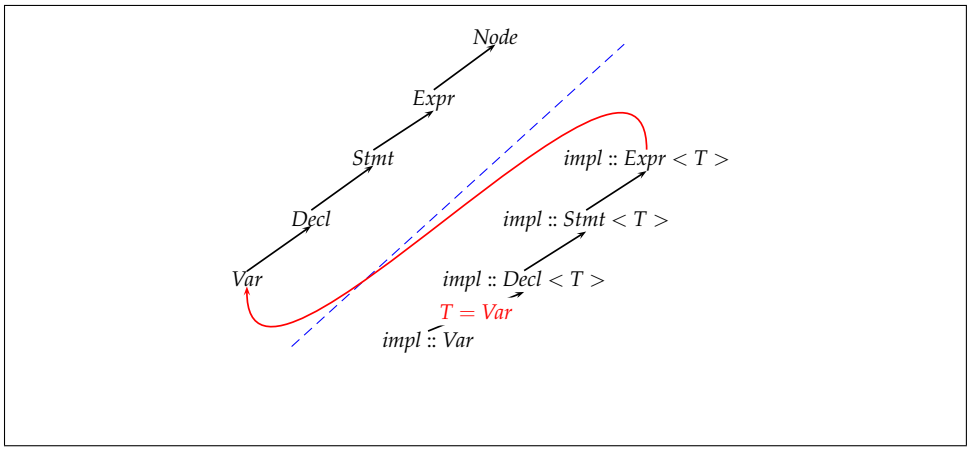

Figure 3: Current design of the IPR library

specific interface class is then derived from the appropriate structural class template (`Unary`, `Binary`, or `Tertiary`). For instance, an array type is structurally a binary type expression and is therefore represented by node with the following IPR interface:

```
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.

5.2.2. 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.

5.2.3. Examples. The IPR does not consider C++ built-in types, such as `int`, special. Rather, it represents them in a way that allows uniform treatment of all types (user-defined and built-in). To get specific about the properties of built-in types, we need knowledge of the target machine's model (e.g., the size of `int`, its alignment, etc.). However, those specific details are not needed for a high-level representation of a program. Currently, the IPR provides only a partial interface to such compiler- and machine-specific information. For example, we can acquire the information needed to evaluate constant expressions, but not answer questions about alignment. This interface will be expanded as needed.

To build a node to represent `int`, we first build an *Identifier* node with the name of the type: `"int"`. That *Identifier* node is also an expression. Then, we state that it actually is a type using the type constructor *As_type*:

$$\mathit{As\_type}(\mathit{Identifier}(\texttt{"int"}))$$

In IPR *every* node has a type. But what kind of type is this `"int"`? IPR represents the notion of a "type of a built-in type" as a `"typename"` node:

As_type(*Identifier*(`"typename"`))

Finally, we state that the `"typename"` node is the type of the `"int"` node. In all the code looks like this:

```
impl::As_type* inttype = unit->make_as_type(unit->get_identifier("int"));
inttype->constraint = unit->get_typename();
```

Since this operation is done frequently (at least for all the 20 built-in types), the operation is abstracted into dedicated member functions of `impl::Unit`:

```
const ipr::As_type& get_as_type(const ipr::Expr&);
const ipr::As_type& get_as_type(const ipr::Expr&, const ipr::Linkage);
```

The second version constructs types with specified linkage (the default being C++ linkage). These functions are what we expect users to call in the situation we just described. The result is this set of nodes:

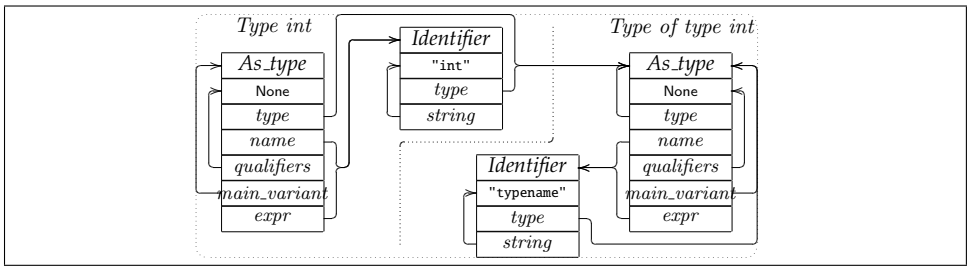

Figure 4: IPR model for the C++ type `int`

The conventions for our node diagrams are:

1. all nodes are drawn as boxes, labeled with their node constructor names;
2. boxes contain slots for the names of supported operations. Arrows indicate virtual function calls;
3. values like sequence sizes or cv-qualification constants are depicted directly in slots.

The `As_type` node is where the name of a type is tied to the properties of the type. A given `As_type` may contain data specifying properties of a type, such as the size and operations applicable to an `int` or the concept of a template argument type (as proposed for C++0x [9, 15]). We don't actually need to add elaborate data to `As_type` nodes; in some cases, we have found it more convenient to keep tables of properties "elsewhere" and find them using the `As_type` as a lookup key.

For type `int`, this representation may seem over-elaborate. However, it allows for the essential uniform treatment of all types, which significantly simplifies use and it imposes no significant cost. The notion of a concept [9, 15] is essential for template arguments, where "what is the type of this type?" or "what are the requirements for this set of types and values?" become central questions in most

high-level analysis. Answering such questions simply and efficiently for every type is key to semantics-based analysis and transformations.

`As_type` provides `qualifiers()` and `main_variant()` operations. Like most IPR operations, these operations directly reflect C++ semantics. For example, for the type `const int`, `qualifiers()` returns `const` and `main_variant()` returns `int`.

To build a node that represents a pointer to `int`, we start with the representation of `int` and apply the *Pointer* type constructor:

```
unit->get_pointer(inttype);
```

That way, we get:

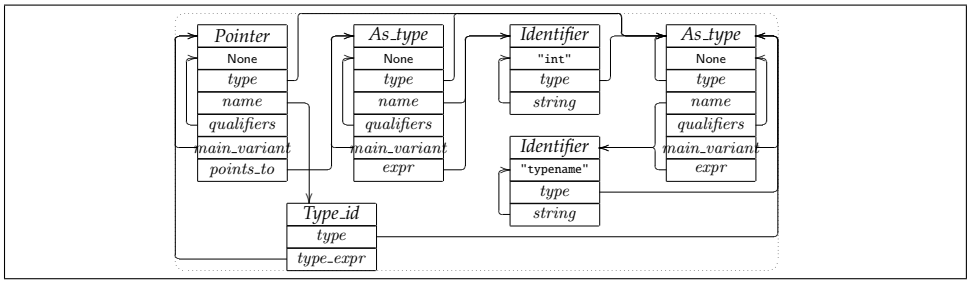

Figure 5: IPR model for the C++ type `int*`

What is the name of `int*`? The name is a *Type_id* node. All nodes that represent types support the operation *name* , providing a uniform way of accessing every kind of name. The `name()` operation returns an expression that represents the name of a `Type`. In this example, `name()` returns the `Identifier("int")` for `int` and the `Pointer` to `int` for `int`. Again, this exactly mirrors the ISO C++ standard where we can talk of types with no conventional names such as `int*&` and `struct{int a; char* p; }`.

A *Type_id* node supports the operation *type_expr* such that

$$type\_expr\,(Type\_id\,(t)) = t,$$

and as a node that represents an expression, its type satisfies the relation

$$type\,(Type\_id\,(t)) = type\,(t)\,.$$

5.3. 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* [12]. 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`.

5.4. Effects of unification

Building nodes, without node sharing, is very simple: allocate enough storage to store the node and set its components. The obvious expense is wasted memory. The representation of the type `int`, for instance, requires $6+2x$ words for *As_type*; and $3 + x$ words for *Identifier*, excluding storage for the string `"int"` – where x designates allocation overhead (usually 2 words). So, that representation uses at least $9 + 3x$ words. In that account, we do not include the storage needed to represent the concept of type, as we take it to be shared by most types. Therefore, the representation of the type of `copy` (§2), with no sharing, needs at least $36+12x$ words for the four occurrences of `int`. On popular machines where a word is 4 bytes and allocation overhead is at least 2 words, that representation needs at least 240 bytes.

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`s. 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. The first author of this paper was a GCC maintainer and shipping manager.

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 to decide which nodes need attention". With node sharing, those comparisons are simple pointer comparisons. Without node sharing they are recursive double dispatch operations. This difference in run-time cost is greater than a factor of 100: a single machine instruction comparing two words v.s. two indirect function calls. The time and space 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.

Obviously, the idea of node sharing (unification) is not new. However, in the context of program transformation, we see it as more than an optimization technique. It is essential for scaling beyond toy programs and toy examples, but 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.

6. Dealing with Real-world Complexities

"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 many more elements (both standard and non-standard) have been added since. This seriously complicates the design of a complete representation for C++. On the one hand, we must save the developers of tools from this incidental complexity ("abstract them away") wherever possible. On the other hand, these "details" must be dealt with to produce a tool, rather than a toy and sometimes tool developers are specifically interested in such messy details. For example, we have experimented with tools to help "rejuvenate" code by replacing older idioms and language features with more modern ones[25]. C++ is complicated. However, much of that complication is the direct result of almost three decades of serious industrial use. Other languages in wide use have suffered similar increases in size, complexity, and diversity of styles of use. It follows that the techniques we have developed for C++ are likely to be applicable a broad range of real-world languages with long-lived code bases and dialects. When looking at the effort needed, we 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.

6.1. 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 surprisingly 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 have a different *home scope*. 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.

In the IPR, 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;
   // ...
};
```

If a name appears only in one scope, its `home_region()` is the same as its `lexical_region()`.

6.2. 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); }        // invoke print(double)

void print(int);

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

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

The IPR represents different functions with the same name in the same scope as overload sets; different declarations of the same function are linked to the first declaration of that function. All declarations are placed in their proper scopes and their proper places in those 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 IPR `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.

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

6.3. Lowering

Even at the level of an AST, different users want different levels of representation. Replacing constructs present in the original source code with equivalent constructs to gain a simpler and more uniform representation is called "lowering." Macro expansion, elimination of type aliases, and putting expressions on a common form (e.g. by replacing operators such as `++` and `->` with alternatives) are examples of "lowering." Making type conversions, constructor and destructor calls explicit can be considered "lowering," as can expansion of `#include`d header files,

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 for particular examples of lowering, such as elimination of `typedef` names, it would be impractical to provide every desired combination of features to be "lowered." The IPR provides a few common example and users have developed IPR-to-IPR tools for more specialized lowering needs.

6.3.1. Macros. 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 is always ideal. However, we did not have much choice. Macros are inherently irregular, so that distinctions among fundamental notions — such as, declaration, statement, and expression — are often blurred by macros.

IPR's inability to represent macros is fundamental, but also a major problem for source-to-source transformation applications such as source code rejuvenation

[25]. Consequently, we are working on a tool to classify macros and replace "well behaved macros" with better-behaved language features, such as `const`s, `inline` functions, and templates. Such replacement is itself a form of source code rejuvenation.

6.3.2. Type aliases. Before lowering, IPR takes 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 that assign meaning to typedef names (and to other aliases). For example, this can be used to detect inconsistent programming styles that may hide bugs from a human reader. "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.

6.3.3. Canonical use of operators. Several operations can be achieved by different uses of operators. For example, `++x` and `x+=1` are equivalent and may be equivalent to `x.operator++()`. Consequently, there is the issue of how to use of such operations: Sometimes we want to see what the programmer wrote and sometimes we would prefer a canonical representation. 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.

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

Now, 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)`.

6.3.4. Header file inclusion. Some of the most complicated code you will ever see is code that most programmers do not usually see: The contents of header files, especially the contents of header files related to systems interfaces. For complete and precise analysis, we have to be able to handle such code (and we do, see §6.4), but to simplify analysis we sometimes want to treat the declarations in a header as a set of primitives (assuming their definition is correct and need not be part of the analysis) and for program-to-program transformation it is typically essential that the generated program still `#include`s a header rather than containing the contents of the header.

By default, we expand headers, but that can lead to surprises as an innocuous header, such as the ISO standard library `iostream`, can drag in tens of thousands of lines of code (mostly because it itself `#include`s files full of implementation details). Consequently, the use of "skeletons" is becoming popular. In this context, a "skeleton" is a simplified header file containing only what a standard requires and no messy implementation details. In particular, a "skeleton" contains no types used only to specify implementation details, no functions except the ones that are part of the documented interface, and no data members of classes. "Skeletons" for the standard library headers can be extracted from the ISO standard itself, but for other libraries and popular operating systems headers they must be generated (by hand or by an IPR tool) from the source code itself. The IPR user must then choose chose between "skeletons" and the real headers.

Unfortunately, a program cannot fully automate the generation of "skeletons." If our aim is portability, we still need to (by hand) eliminate non-standard additions to the contents of header file. For example, `strdup()` is not part of the ISO C standard library even though it is often found in `<string.h>`.

6.4. 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 do not represent "details" of `<iostream>` in the IPR unless the user explicitly requests it (see §6.3.4).

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.

6.5. 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: Compile all source files one at a time and store their XPR representation. Then add the XPR files that you want to a single IPR 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. We can use the "skeleton approach" §6.3.4 or we could determine the use of headers as we go along. 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.

6.6. 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 number of distinct node types is 157. This count excludes nodes representing vendor-specific, non-standard features. Of these 157 nodes, 68 nodes represents individual C++ operators (such as `+` and `*`) and 20 nodes represents individual built-in types (such as `int` and `long double`). We represent individual operators as separate node types, rather than as a single node with a operator-type field so that we can select among them using a virtual function call rather than as switch-statement. Had we chosen to minimize the number of nodes rather than the complexity of user code, the number of nodes would have been 71 or lower.

The code for IPR is available from the authors.

6.7. Traversal

Traversal of IPR nodes is based on a combination of the Visitor Design Pattern and Generic Programming techniques. As outlined earlier the IPR library offers two sets of classes:

1. a purely functional, that support only non-mutating operations; and
2. a set of implementation classes supporting both mutating and non-mutating operations.

The immutable classes are useful for the majority of users. They support analysis and the generation of new code from an unchanged source. The mutating classes

are more specialized, harder to use well, and geared toward applications that need to change nodes in place. Given node sharing, "mutating" really means "make a new node and replace an old node with it", rather than indiscriminately writing to node data members. We illustrate uses of both sets with two simple problems developed in the next sub-sections.

6.7.1. Simple traversals. Consider the problem of traversing a program, looking for all statements of the form

```
a = a + b;
```

collecting the assignment expression and the left-hand-side sub-expression (which can be a simple variable or more complicated expression).

To solve that problem, we must visit the body of all function definitions, starting from the global namespace, walking down every class or function definition. The expressions are collected in form of a map

$$\textit{assignment expression} \mapsto \textit{left hand side}.$$

The corresponding traversal code is straightforward. First, we need a visitor class that will visit only the interesting nodes, and do nothing to others. The IPR universal `Visitor` base class is structured in a way that it provides a default hook for acting on interface nodes (e.g. `Var`, `Block`, `Assign`, `Array`) assuming that we know what to do given their general classification (e.g. declaration, statement, type, expression) That is, the `Visitor::visit` member functions for all node types other than `Decl`, `Stmt`, `Expr`, and `Type` are implemented as forwarding functions to one of

```
struct Visitor {
   // ...
   virtual void visit(const Decl&) = 0;
   virtual void visit(const Stmt&) = 0;
   virtual void visit(const Type&) = 0;
   virtual void visit(const Expr&) = 0;
   // ...
};
```

Those functions are pure virtual, therefore must be overridden in derived classes.

For lot of simple visitor classes, it can be a tedious exercise to have to provide "dummy" overriding definitions for those `visit` member functions. Consequently, the IPR library has "helper" visitors, such as the `Constant_visitor` template that applies the same function object to all nodes. For example `Constant_visitor<No_op>` is the visitor class that does nothing to all kind of nodes.

Given that, the definition of the gathering function is simple:

```
using namespace std;
using namespace ipr;

// Collect { a = a + b } in this container
typedef map<const Assign*, const Expr*> Map;
```

```
struct Gatherer : Constant_visitor<No_op> {
   Map result;    // hold matching expressions

   // an assignment-statement is a good place to search
   void visit(const Assign& e) {
      if (const Plus* x = view<Plus>(e.second()) {        // assigns a sum
         const Expr& lhs = e.first();
         if (lhs == x->first())          // assigns to left-hand operand of+
            result[&lhs] = &x->first();
      }
   }

   // for an expression statement, the contained
   // expression might contain an assignment
   void visit(const Expr_stmt& e)
   {
      e.expr().accept(*this);
   }

   // search every statement in a block
   void visit(const Block& b)
   {
      const Sequence<Stmt>& stmts = b.body();
      for (int i = 0; i < stmts.size(); ++i)
         stmts[i].accept(*this);
   }

   // ...
};

void summary(const Unit& unit)
{
   const Sequence<Decl>& decls = unit.get_global_scope().members();
   Gatherer gatherer;

   for (int i = 0; i < decls.size(); ++i)
     decls[i].accept(gatherer);
   print_matches(gatherer.result);
}
```

The function `print_matches` print out a summary of all expressions that match our criteria and stored in `gatherer.result`.

6.7.2. Simple transformations. Once we have found expressions that matches the pattern

```
x = x + y;
```

we are interested in replacing them with the equivalent form

```
x += y;
```

on the condition that there is an operator +=, either built-in or user-defined, of the appropriate type accepting a modifiable lvalue (of the same type as `x`) as its first operand, and `y` as its second operand. This transformation requires that we have complete type information about the declarations in scope. Consequently, it is not (just) a syntactic transformation. The building block of that transformation is the following node-building function

```
// transform ``a = a + b'' to ``a += b'',
// setting the type ``t'' on the resulting expression.
const Plus_assign*
mutify(impl::Unit& unit, const Assign& assignment, const Type& t)
{
   const Expr& lhs = assignment.first();
   const Expr& lhs = assignment.second();
   impl::Plus_assign* result = unit.make_plus_assign(lhs, get_second(rhs))
   result->constraint = &t;
   return result;
}
```

The rest of the transformation consists of cloning nodes not in our `gatherer.result` table, and replacing those in the tables by their images. Again, this clone-and-substitute operation is performed by an appropriate visitor and has structure similar to what we discussed in the previous section.

A further transformation is to replace all statements of the form

```
x = x + i;
```

into

```
++x;
```

when `i` is an integral constant expression with value 1. Doing this requires not only sufficient understanding of the C++ code to recognize constant expressions, but also the ability to evaluate them. IPR provides that, so that the application programmer can write simple code that depends on values. The `x=x+1` to `++x` transformation has implicit and useful semantics implications: it increases the generality of an algorithm when `x` is an iterator, *i.e.* it transform an operation that requires a random-access iterator into an operation that assumes only forward iterator property.

7. 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, 29, 30, 2, 22, 28, 21] targeting static analysis and transformations of C++ programs. For example, CodeBoost [2, 3, 20] focuses on transformations of C++ programs, for numerical PDE solvers, written in the Sophus style. Simplicissimus [29, 30] and ROSE [28] are other projects for transforming C++ programs. Coverity is a widely used and

very complete commercial set of C and C++ analysis tools [4]. Many of these systems are commercial and not documented in the literature. Few aim to handle full Standard C++, few aim at generality (as opposed to specific applications), and few aim at compiler independence. None — to our knowledge — aim at all three.

The IPR is designed to fully support the next generation of ISO standard C++ (nicknamed C++0x)[19, 34]. The basic supports for that are in place, though we will have to wait for the final standard and compiler support to complete this work.

Obviously, our immediate aims include applications that further test the generality and portability of the IPR and its associated tools. We have been able to represent the full source code of FireFox. We have simple visualization tools and several experiments related to the analysis and simplification of code have been and is being conducted. For example, we have used IPR in experiements that extracted concepts (template argument requirements) from C++ template definitions [27], detected and analysed loops for potential simplification [26], and as part of the infrastructure of for abstract interpretation of C++ programs. To ease the development of such programs, we have developed an IPR based pattern-matching language and library[26], and library support for selective traversal [35]. We are working on experiments 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. Experience with students indicate that we also need to either provide simpler ways to specify simple traversals and transformations or to better explain what IPR offers.

From the standpoint of the structure of the IPR, the most important direction of work is to more systematically handle modularity and lowering.

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.

8. 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

[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] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World. *Communications of the ACM*, 53(2), February 2010.

[5] clang: a C language family frontend for LLVM. `http://clang.llvm.org/`.

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

[7] Gabriel Dos Reis and Bjarne Stroustrup. The Pivot. `http://parasol.tamu.edu/pivot`.

[8] Gabriel Dos Reis and Bjarne Stroustrup. A Formalism for C++. Technical Report N1885=05-0145, ISO/IEC SC22/JTC1/WG21, July 2005.

[9] 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.

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

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

[12] A. P. Ershov. On programming of arithmetic operations. *Commun. ACM*, 1:3–6, August 1958.

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

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

[15] 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.

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

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

[18] J. Järvi, B. Stroustrup, and G. Dos Reis. Decltype and Auto (revision 4). `http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2004/n1705.pdf`, September 2004. ISO/IEC JTC1/SC22/WG21 no. 1705.

[19] ISO/IEC JTC1/SC22/WG21. Programming Languages C++. Technical report, ISO, March 2010. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3092.pdf.

[20] 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.

[21] 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.

[22] 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.

[23] John C. Mitchell. Type systems for programming languages. pages 365–458, 1990.

[24] 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/`.

[25] Peter Pierkelbauer, Damian Dechev, and Bjarne Stroustrup. Source Code Rejuvenation is not Refactoring. In *36th international conference on current trends in theory and practice of Computer Science*, January 2010.

[26] Peter Pirkelbauer. *Programming Language Evolution and Source Code Rejuvenation*. PhD thesis, Texas A&M University, September 2010.

[27] Peter Pirkelbauer, Damian Dechev, and Bjarne Stroustrup. Supporr for the Evolution of C++ Generic Functions. In *Software Language Engineering*, volume 6563 of *Lecture Notes in Computer Science*, pages 123–142. Springer Berlin / Heidelberg, October 2010.

[28] 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.

[29] 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.

[30] 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.

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

[32] Bjarne Stroustrup. A History of C++: 1979-1991. In *Proceedings of ACM Conference on History of Programming Languages (HOPL-2)*, March 1993.

[33] Bjarne Stroustrup. Evolving a Language In And for the Real World: C++ 1991-2006. In *ACM HOPL-III*, San Diego, California, June 2007. ACM Press.

[34] Bjarne Stroustrup. What is C++0x? *CVu*, 21, 2009. Issues 4 and 5.

[35] Luke A. Wagner. Traversal, Case Analysis, and Lowering for C++ Program Analysis. Master's thesis, Texas A&M University, June 2009.

Gabriel Dos Reis
Texas A&M University,
College Station, TX-77843, USA
e-mail: `gdr@cse.tamu.edu`

Bjarne Stroustrup
Texas A&M University,
College Station, TX-77843, USA
e-mail: `bs@cse.tamu.edu`