

Representing C++ directly, completely and efficiently

G. Dos Reis B. Stroustrup

Abstract

We describe a systematic representation of C++ for complete semantic analysis and semantics-based transformations. In particular, we describe how general unification is key to compact representation, fast type-safe traversal, and scalability. The tradeoffs between the complexity of unification and the complexity of comparison are discussed and illustrated by a few measurements. The representation is designed to be general enough to handle likely C++0x extensions that affect the type system such as `decltype` and concepts.

1 Introduction

The C++ programming language [3, 4, 11] has been, for the last two decades, adopted in wide and diverse application areas¹. Beside traditional applications of general purpose programming languages, it is being used in embedded systems, safety-critical missions, air-plane and air-traffic controls, space explorations, etc. As a consequence, the demand for static analysis and advanced semantics-based transformations of C++ programs is becoming pressing. This paper discusses a complete, efficient and direct representation of C++ in C++, as designed for *The Pivot* infrastructure.

The Pivot is a framework for static analysis and transformation of C++ programs, being developed at Texas A&M University. In particular, it aims at the support for high-level parallel and distributed programming techniques. It “understands” the higher levels of C++ (e.g. templates, specializations, concepts), crucial for advanced optimizations, validation of safety, enforcement of dialects, support for staging libraries. *The Pivot* infrastructure consists in:

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;

¹see <http://www.research.att.com/~bs/applications.html>

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 discussion of the *The Pivot* infrastructure will be found elsewhere. Our main topic, in this paper, is the description of the design principles of the Internal Program Representation component.

We are currently able to represent all of C++ and we can generate IPR for almost all of C++ from two compilers (GCC/g++ and EDG front-ends.) We expect to be able to do experiments using whole translation units of real-world C++ programs within months.

This document is organized as follows: the design principles for the IPR are explored in §2; then we look at representations of the C++ type system in §3; sections §4 and §5 discuss node equivalence, benefits, costs and complexity of unification; unification rules are studied in §6.

2 Design rules

The goals set for *The Pivot* directly guide the design criteria of IPR. It aims at the support of all C++, with the exception of macros. Our ideals are that IPR should

- be *complete* — it should represent all Standard C++ constructs, but not macros before expansions, not other programming languages;
- be *general* — not targeted to a small area of applications; it should be useful to the wide C++ community;
- be *regular* — it should contain full C++, but not mimic irregularities. In particular, general rules should be preferred to long lists of special cases;
- emphasize *types* — types give meaning to programs and IPR can be thought of as a fully typed abstract syntax tree;
- be *compiler neutral* — it should not be tied to a particular compiler details or implementations;
- be *efficient and elegant* — it should be able to handle hundreds of thousands of lines of code on common machines (such as our laptops.)

Within IPR, C++ programs are represented as sets of graphs. We'll discuss efficient and scalable representations directed by the idea of node sharing, *i.e.* consistent reuse of nodes based on syntactic equivalence and semantic equivalence. For example, consider the declaration

```
int* copy(const int* begin, const int* end, int* out);
```

that declares a function `copy`. Its representation needs creations of nodes for the various entities involved, e.g. function-declaration, parameters. All of those entities have types, and there is a visual pattern. Information and assumptions that are implicit in the C++ syntax are rendered explicit in the IPR model. For example, that declaration does not explicitly mention an exception specification; so the C++ semantics are that the function `copy` may throw an exception of any object type. For completeness, its IPR model contains an exception specification referring to ellipsis (`...`), the node that represents the notion of “any object type”. The C++ types `int*` and `const int*` are mentioned twice, respectively. For the internal representation purpose, the issue immediately arises as whether four distinct nodes should be built to represent them or whether previously constructed nodes should be reused consistently, *i.e.* shared. While the discussion is primarily driven by nodes that represent types, it will be clear that most conclusions are also applicable to nodes representing C++ expressions.

3 Type expressions

This section is devoted to the syntactic representations and semantics of the C++ type system. We discuss those representations as C++ library, and diagrams. Finally, we examine some examples to get a feeling of the fundamental issues.

The syntactic representations are important in order to accurately capture template definitions. However, it should be emphasized that the representations offered by the IPR library is not just about the syntax of C++ entities. Semantics notions such as overload-sets, scopes are fundamental parts of the library. It is also prepared for integration of concepts [14, 15, 12], and support for concept-based analysis and transformation.

3.1 Syntax and semantics

The IPR language (the language defined by IPR nodes) consists mostly of expressions. These are generalizations of Standard 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. Classic expressions, statements and declarations are discussed in §6.7. Only the type system representation is considered here.

Types are defined by the following rules

$$\begin{aligned}
\text{type} ::= & \text{Pointer}(\text{type}) \mid \text{Reference}(\text{type}) \mid \text{Array}(\text{type}, \text{expr}) \\
& \mid \text{Const}(\text{type}) \mid \text{Volatile}(\text{type}) \\
& \mid \text{Function}(\text{product}, \text{type}, \text{sum}) \\
& \mid \text{Class}(\text{scope}, \text{scope}) \mid \text{Union}(\text{scope}) \mid \text{Enum}(\text{scope}) \\
& \mid \text{Namespace}(\text{scope}) \\
& \mid \text{Decltype}(\text{expr}) \mid \text{Type_expr}(\text{expr}) \\
& \mid \text{Template}(\sigma_{\text{param}}, \text{type}) \\
& \mid \text{Product}(\sigma_{\text{type}}) \mid \text{Sum}(\sigma_{\text{type}})
\end{aligned} \tag{1}$$

where the variables expr , type , product , sum , scope , σ_{param} , σ_{type} range over generalized expressions, generalized types, product types, sum types, scopes, sequences of parameters, sequences of types, respectively. The following sub-sections describe the meaning of each of these rules and their motivations.

3.1.1 Pointer, reference and array type expressions

The type constructors *Pointer*, *Reference* and *Array* correspond to the usual operations of constructing pointer types, reference types, array types out of C++ types. Those type constructors are fairly conventional. The operations *points_to*, *refers_to*, *element_type* and *bound* extract the arguments used to construct pointer, reference or array types according to the identities

$$\begin{aligned}
\text{points_to}(\text{Pointer}(t)) &= t, & \text{refers_to}(\text{Reference}(t)) &= t, \\
\text{element_type}(\text{Array}(t, e)) &= t, & \text{bound}(\text{Array}(t, e)) &= e.
\end{aligned}$$

3.1.2 CV-qualified type expressions

The cv-qualifiers `const` and `volatile` are represented by the type constructors *Const* and *Volatile*, respectively. Unlike other type constructors, these type constructors retain essential properties of their arguments. For example, given a pointer type t , the type expression $\text{Const}(t)$ is still a pointer type. Similarly, if t is an integer type, then $\text{Volatile}(t)$ is also an integer type. The cv-qualifier type constructors enjoy a commutation relation:

$$\text{Const}(\text{Volatile}(t)) \rightsquigarrow \text{Volatile}(\text{Const}(t))$$

Those properties suggest the operations *main_variant* and *qualifiers* for all types. The operation *main_variant* yields the cv-unqualified version of its argument. The value of the operation *qualifiers* is any of the four symbolic constants: `none`, `const`, `volatile` and `const+volatile`. The value `none` indicates that a type is not the result of any cv-qualifier type constructor. If t is

constructed by *Const*, then the value of *qualifiers(t)* is *const*. Similarly, *qualifiers(t)* is *volatile* if and only if *t* is the result of *Volatile*. Finally, if *t* is the result of *Const* followed by *Volatile* or the other way around, thanks to commutation, the value of *qualifiers(t)* is *const+volatile*. The operations *qualifiers* and *main_variant* are related by the identity

$$\mathit{qualifiers}(\mathit{main_variant}(t)) = \mathit{none}.$$

3.1.3 Function type expressions

The type constructor *Function* corresponds to the usual notion of function type in C++. It takes the parameter-type list, the return type and the exception specification as operands. The arguments used to construct a function type can be extracted by the operations *source*, *target* and *throws* respectively, and that according to the rules

$$\begin{aligned} \mathit{source}(\mathit{Function}(s, t, x)) &= s, \\ \mathit{target}(\mathit{Function}(s, t, x)) &= t, \\ \mathit{throws}(\mathit{Function}(s, t, x)) &= x. \end{aligned}$$

It should be observed that Standard C++ does not formally make exception specifications part of function types. However, the IPR approach is to preserve information and enable style-checker tools that, for instance, enforce static verification of exception specifications.

3.1.4 User-defined type expressions

The usual user-defined types (enums and classes) are represented by the type constructors *Enum*, *Class* and *Union*. The IPR component regards namespaces as user-defined types; they are constructed with the type constructor *Namespace*. All user-defined types take sequence of the members they declare as a scope operand. The operand can be retrieved by the operation *scope*

$$\begin{aligned} \mathit{scope}(\mathit{Enum}(s)) &= s, & \mathit{scope}(\mathit{Class}(b, s)) &= s, \\ \mathit{scope}(\mathit{Union}(s)) &= s, & \mathit{scope}(\mathit{Namespace}(s)) &= s. \end{aligned}$$

The *Class* type takes an additional operand which is the scope of base class declarations. The scope of base classes can be recovered by the operation *base_scope*:

$$\mathit{base_scope}(\mathit{Class}(b, s)) = b.$$

In principle, one could conceive of notion of base type for all user-defined types. For example, for enums, it would specify the underlying type — which in current C++ is an integer type, precisely which is left as a compiler implementation detail. For namespaces, it would correspond to some form of using-directive. However, we do not pursue that generalization here.

3.1.5 Declaration type and named type expressions

The *Decltype* constructor is a facility to query the type of an expression (see [6]). The type constructor *Type_expr* turns an arbitrary expression into a type. A type constructed by *Decltype* or *Type_expr* supports the operation *expr* which yields the argument used to construct that type:

$$\text{expr}(\text{Decltype}(e)) = e, \quad \text{expr}(\text{Type_expr}(e)) = e.$$

The type query functionality is not part of Standard C++ but is one of the key features of C++0x to support generic programming. The *Type_expr* facility is needed to introduce built-in types (see §3.2) or type variables and to handle dependent types in template declarations.

3.1.6 Product type and sum type expressions

Product and *Sum* types do not exist directly in C++. However they are useful notions informally used by C++ programs. The *Product* type constructor is currently used to represent parameter-type lists of functions. It is literally the Cartesian product of the types used in the parameter-type list. The *Sum* type constructor is dual to the notion of *Product*. It represents a collection of types that support a common set of operations. For example, in function exception specification, it means that the function may throw any exception of type named in that sum. A *Sum* type can also be used to support the idiom where a collection of classes (deriving from a common abstract base class) implements the notion of type-safe or discriminated union.

Both *Product* and *Sum* supports the subscription and *size* operations. A subscript with an integer value refers to the type element at that position in the argument used to construct the product or sum type. The operation *size* reports the number of types in the product or sum.

$$\begin{aligned} \text{size}(\text{Product}(s)) &= \text{size}(s), & \text{size}(\text{Sum}(s)) &= \text{size}(s), \\ \text{Product}(s)_i &= s_i, & \text{Sum}(s)_i &= s_i. \end{aligned}$$

3.1.7 Template type expressions

Finally, IPR takes the approach that “template” is a type, constructed with *Template* taking its parameter-type list and the type expression being parameterized. Notice that this allows for any declaration to be parameterized, including variable and namespace declarations. The IPR aims for greater generality and regularity than what C++ currently offers.

Given a template type, one can retrieve its parameters and the type being parameterized with the operations *parameters* and *parameterized*:

$$\text{parameters}(\text{Template}(p, t)) = p, \quad \text{parameterized}(\text{Template}(p, t)) = t.$$

The *parameterized* is also known, in the Standard C++ definition text, as the *current instantiation*. For example, in the declaration

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

the current instantiation is the class-expression

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

where the template-parameters are used to specify the class-body.

3.2 Representations

3.2.1 As C++ library

The IPR library provides users with a complete set of classes to cover all aspects of Standard C++. Those classes are designed using well-established object-oriented principles. They are organized in hierarchies, and can be divided into two major categories:

- abstract classes, for the interface to the representations; and
- implementation classes, for the concrete representation.

The interface classes support only non-mutating operations. All the operations mentioned in the preceding sections are present in the interface classes (as virtual functions). Traversals are based on the Visitor Design Pattern.

IPR is designed to yield information in the minimum number of indirections. IPR indirections are all semantically significant. That is an indirection refers to 0, 2 or more different types of information, but not 1. 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 [5]. Virtual functions are also preferable for simplicity, code clarity, and maintenance. Should an indirection become a performance bottleneck, we will consider manually unrolling its function call, but that is only a last resort.

Interfaces. As can be seen from the rule (1), type expressions and classic expressions are the result of unary, or binary, or ternary node constructors. Only the type of the arguments and the name of the supported operations differ. But the abstract structures are, however, similar. Consequently, we have introduced three class templates to capture those commonalities. For example:

```

// -- Binary<> --
template<class Cat = Expr, class First = Expr, class Second = Expr>
struct Binary : Cat {
    typedef Cat Category;
    typedef First Arg1_type;
    typedef Second Arg2_type;

    virtual const First& first() const = 0;
    virtual const Second& second() const = 0;
};

```

The template-parameter `Cat` indicates the category of node constructor – either (classic) expression, type, statement or declaration. That category, for most node constructor, is that of expressions, hence the default value to `Expr`. The remaining template-parameters indicate the type of the arguments expected by the node constructor. Most node constructors expect general expressions; which explains the default values.

An interface class derives from the corresponding structural class template and forward its operations to the abstract selectors of that class template. All type constructors, except *Const* and *Volatile*, are represented by a class, bearing the same name. For example:

```

// -- Function --
struct Function : Ternary<Type, Product, Type, Sum> {
    const Product& signature() const { return first(); }
    const Type& target() const { return second(); }
    const Sum& throws() const { return third(); }
};

```

Concrete representations. IPR concrete classes are instantiations of the implementations classes for the structural class template. For example:

```

// -- Unary_factory<> --
template<class Interface, class Cat_impl = Expr_impl<Interface> >
struct Unary_factory {
    typedef typename Interface::Arg_type Arg_type;
    Cat_impl* make(const Arg_type*);
    Cat_impl* make(const Arg_type*, const Type*);

    struct Impl : Cat_impl {
        struct Comparator {
            bool operator()(const Impl&, const Impl&) const;
        };
        explicit Impl(const Arg_type* o) : rep(o) { }
        const Arg_type& operand() const { return *rep; }
    private:
        const Arg_type* rep;
    };
};

```

```
private:
    std::set<Impl, typename Impl::Comparator> data;
};
```

The template-parameter `Interface` stands for the interface of node constructor for which the factory will be instantiated. The second template-parameter `Cat_impl` specify the implementation of the common operations of interface's category. `Type_impl<Interface>`, for instance, implements the common operations *qualifiers* and *main_variant*. Since most node constructors are expressions, the template-parameter `Cat_impl` defaults to `Expr_impl<Interface>`. For example, nodes that represent pointer types are built with (a class derived from) the factory

```
Unary_factory<Pointer, Type_impl<Pointer> >
```

The IPR library takes care of memory management, so that users don't have to solve that problem too.

3.2.2 As diagrams

We find it very useful to use diagrams when talking about the IPR representations of C++ constructs either in their abstract forms as found in §3.1 or their representation as C++ classes. The conventions used in this paper are

- all nodes are drawn as boxes, labeled with their node constructor names;
- boxes contain slots for the names of supported operations. An arrow based on a named slot indicates a virtual function call;
- some values like sequence sizes or cv-qualification constants are depicted directly in slots.

3.2.3 Examples

For illustration of the core issues of node sharing and broader view of notions involved in IPR, we will walk through the basic steps of how nodes are built to represent types in the example from the introductory section.

The IPR library does not have a particular knowledge of the C++ built-in types. Rather, it represents them in a way that is both suitable for user interpretations in specific contexts, and general enough to work with other similar situations, e.g. dependent types in template declarations or *concepts* [14, 15, 12] (when they are integrated into IPR.). It makes minimal commitment as to what they mean. To get more specific about the properties of built-in types, one needs knowledge of the target machine's model (e.g. what is the size of the natural integer type, what is its alignment, what

are the extrema of its value set, etc.) Those specific details are not needed for the purpose of building a high-level representation of a program.

Built-in `int`. To build a node that denotes the C++ type `int`, first build an *Identifier* node with the string "int". That *Identifier* node is also an expression. Then, state it is actually a type, through application of the type constructor *Type_expr*. Those steps are very close to the notion of *built-in*.

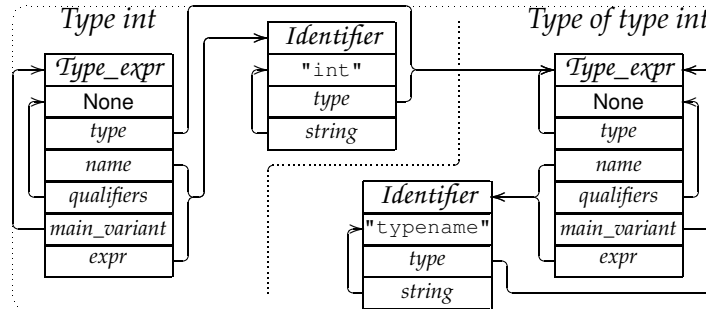


Figure 1: IPR model for the C++ type `int`

From *The Pivot* perspective, every type is an expression, and as such has a type. The type of `int`, is the concept of type. There is a specific IPR node to denote the concept of type; it is represented as

$$\textit{Type_expr}(\textit{Identifier}(\text{"typename"}))$$

and is built similar to the way we just explain for `int`. In this specific case of type `int`, it may seem an over-generalization. However, it imposes no significant cost and it is essential in the context of templates using concepts to constrain template arguments. This is an example of situations where IPR is prepared for likely C++0x extensions and semantics-based analysis and transformations.

The type of an identifier, in general, really depends of the interpretation context or scope. However, in the specific case of `int`, it is a reserved keyword whose meaning cannot be redefined. It always designates the type it stands for. Consequently, the type of *Identifier*("int") is the type of the type it stands for.

Built-in `int*`. To build a node that represents a pointer to `int`, first start with the representation of `int`. Apply the *Pointer* type constructor. Pretty much everything is straightforward. Except, probably, the issue of the name of `int*`. All nodes that represent types support the operation *name*. So what is the name of the type "pointer to `int`"? The answer is it is a *Type_id* node. That provides hooks for code rewriting tools to rewrite source programs according to in-house coding standards. A *Type_id* node supports the

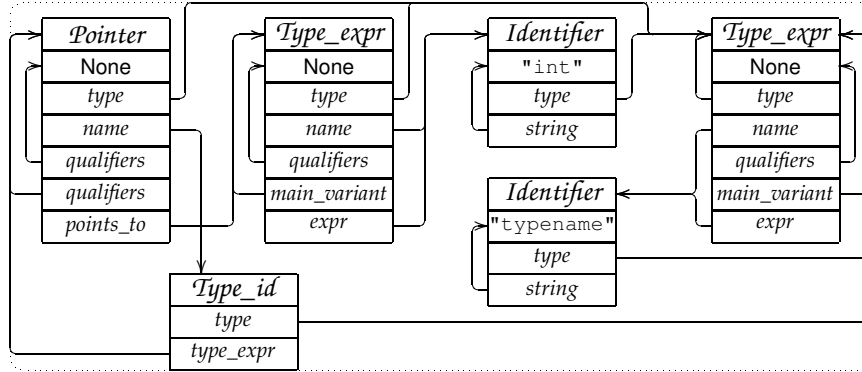


Figure 2: IPR model for the C++ type `int*`

operation $type_expr$ such that

$$type_expr (Type_id(t)) = t,$$

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

$$type (Type_id(t)) = type (t).$$

Type of the copy function. Going back to the original example of the type of the `copy` function. There are four mentions of type `int`, four applications of the `Pointer` type constructor and two applications of the type constructor `Const`. The diagram in figure 3 shows a representation of the type of the function `copy`, with no sharing of nodes. Figure 4, on the other hand, illustrates the situation where the IPR representation uses maximal node sharing. Those two situations are extreme cases. In between, there may be various degrees of node sharing.

4 Node equivalence

IPR nodes are built to represent C++ abstraction, e.g. types, etc. We distinguish between the nodes themselves from what they represent.

4.1 Equality

A node is identified by the address of its storage. So, two nodes p and q are said to be equal when they have the same address, in which case we will write $p = q$.

At least two notions of equality can be defined for C++ abstractions IPR nodes are built to represent:

1. syntactic equality. For instance, two pointer types are syntactically equal if and only if their associated pointed-to types are syntactically equal, and the notion applies recursively;
2. semantic equality. This kind of equality involves Standard C++ language semantic rules. For example, a typedef is semantically equal to the type it is an alias for.

Equality of representations will mostly depend on contexts and uses. Given the wide range target of *The Pivot* infrastructure, it is crucial to clearly separate the two notions of equality of representations. For example, consider the case of a library that uses the notion of color and expresses it as a typedef to `long`

```
typedef long Color;
```

and uses a tool to ensure that calls to functions that manipulate `Color` are made with expressions involving only authorized operations and variables or constants declared with the typedef-name `Color`. Such a tool would need the syntactic equality; the semantic equality would be inappropriate. A variation of that tool may even go and replace the typedef-declaration with a class definition and appropriate function definitions, should the need arise. Conversely, checking whether arguments used to call a function satisfy C++ language rules need the semantic equality.

From now on, when we use *equality of representations* with no other qualification, we imply syntactic equality which we will write as \cong .

4.2 Sharing

Before digging deep into the consequences of node-sharing or absence thereof, let's be more precise about what is meant by node-sharing. By *node sharing*, we mean that two nodes that represent the same entity shall have the same address (or whatever identification assignment is used to designate them). In particular, node sharing implies that if a node constructor is presented twice with equal lists of arguments, then it will yield the same node, as can be seen on figure 4. A node class is said to be *unified* if all nodes of that class consistently feature the sharing property. Since a user-defined type (classes or enums) can be defined only once in a given translation unit, sharing of nodes is already implied by C++ language rules.

The node sharing issue is less straightforward for built-in types and types constructed out of type constructors (including templates), e.g. `int`, `double (*) (double)`, `vector<Shape*>`, etc. That is so because they are not introduced by declarations. They are omnipresent and can be referred to at any occasion. Consequently, node sharing for those types implies maintenance of tables that translate arguments used to constructed node.

Node-sharing through syntactic equivalence has implications on the meaning of *overloaded declarations*, as two declarations might appear overloaded whereas, in fact, 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`. Given consistent naming, useful analysis can be done based on template-arguments names.

5 Life with and without unified nodes

Building nodes, in non-sharing mode, is very simple: allocate enough storage to store the node and set its components. That is so, at the expense of wasted memory. The representation of the type `int` (see §3.2.3), for instance, requires $6 + 2x$ words for *Type_expr*, 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`, with no sharing (see figure 3), 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.

Both notions of equality may be implemented by the same routine, with a parameter indicating whether syntactic or semantic equivalence is in effect. At any case, it requires implementations of the classic “apples-and-oranges” comparison: two double dispatches are needed to recover the types of the nodes being compared, to determine whether they are of the same category and if so apply recursively the same mechanism to their constituent parts. This leads to inefficiency both for node equality and representation equality.

Space is time. It should be obvious that, because nodes are not repeatedly created to denote the same type, node sharing leads to reduced memory usage and less scattering un the address space (and therefore fewer cacher misses.) For example, in the running example of type of the `copy`

function, only one node is created for all the occurrences of `int`, see figure 4.

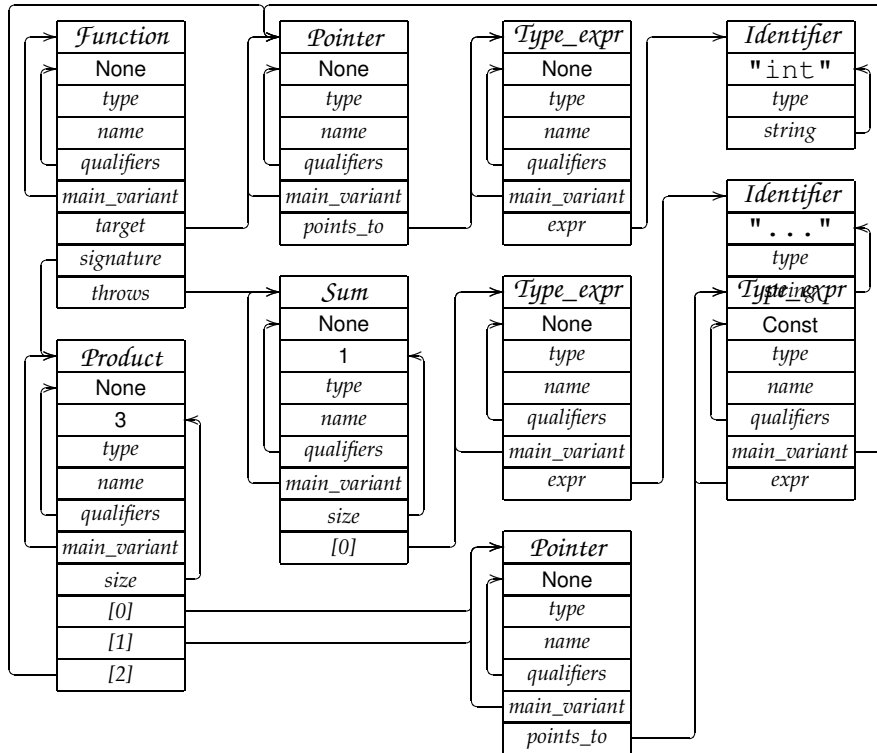


Figure 4: Type of the function `copy`, with maximal node sharing

Experiments with the classic first program

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

based on GCC-3.4.2 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 ($\pm 5\%$). Based on the assumptions and arithmetics from §3.2.3, the duplicate nodes for named types occupy about 2Mb. The hello world program is not a very useful program by itself, but it gives us some quantitative measures. For even medium-sized programs we must expect to 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 consider completely representing multiple translation units simultaneously, unification becomes a critical component of scalability.

For a program analysis that requires type comparison, node sharing offers time efficiency because type comparison is reduced to pointer comparison (see §6). The time efficiency gained by caching computed nodes in lookup tables should be weighted against with the overhead of traversing such tables. Another advantage of node sharing is consistency. Since there is only one integer type named `int`, with a given alignment and size property, there is no need for walking through the whole graph for attaching a given property to each of its representations. That is an important property when merging separately compiled translation units, or doing whole program analysis. The same argument holds for substitutions, that replace uses of a type, say `int []`, with a safe, checked `vector<int>` in a whole translation unit.

6 Syntactic unification of type nodes

From rule (1), it appears that type expressions depend either on expression, or type, or sequence of type, or scopes, or sequence of parameters nodes. The next subsections give a few examples of rules that govern syntactic unification for all categories of nodes that may be involved. Given two nodes p and q , we will write $p \simeq q$ to say that they are unified. From practical implementation point of view, two nodes compare equal (pointer equality) if and only if their representations are unified.

6.1 Pointer, reference and array types unification

Two pointer type or reference type representations are unified if the pointed-to or referred-to type representations are unified. Similarly, two array types are unified if their respective element types are unified and their respective bounds are syntactically equivalent:

$$\begin{aligned} \text{Pointer}(t) \simeq \text{Pointer}(\tau) \quad \text{or} \quad \text{Reference}(t) \simeq \text{Reference}(\tau) &\iff t \simeq \tau, \\ \text{Array}(t, e) \simeq \text{Array}(\tau, \varepsilon) &\iff \left\{ \begin{array}{l} t \simeq \tau \\ e \cong \varepsilon \end{array} \right\}. \end{aligned}$$

Syntactic equivalence of expressions is discussed in §6.7.1.

6.2 Function types unification

Two function type representations are unified if and only if their respective signature representations and their respective return type representations

and their respective exception specification representations are unified.

$$\mathit{Function}(s, r, x) \simeq \mathit{Function}(\sigma, \varrho, \xi) \iff \left\{ \begin{array}{l} s \simeq \sigma \\ r \simeq \varrho \\ x \simeq \xi \end{array} \right\}.$$

The unification of product and sum types is discussed in §6.5.

Example. Consider the following program fragment:

```
void plot(int*);
void plot(int[]);
void plot(int[2]);
```

From syntactic equivalence point of view, these three declarations declare three distinct functions. However, according to Standard C++ rules, they declare the same function — a `void`-returning function named `plot` that takes a pointer to `int` — so their types will be semantically equivalent.

6.3 User-defined types unification

Two user-defined types are unified if and only if their member scopes are equal, and in the case of classes if their base class scopes are equal:

$$\begin{aligned} \mathit{Enum}(s) \simeq \mathit{Enum}(\sigma) \quad \text{or} \quad \mathit{Union}(s) \simeq \mathit{Union}(\sigma) &\iff s = \sigma, \\ \mathit{Namespace}(s) \simeq \mathit{Namespace}(\sigma) &\iff s = \sigma. \\ \mathit{Class}(b, s) \simeq \mathit{Class}(\beta, \sigma) &\iff \left\{ \begin{array}{l} b = \beta \\ s = \sigma \end{array} \right\}. \end{aligned}$$

6.4 Declaration type and named type unifications

Two declaration types are unified if and only if their respective expression operands are syntactically equal; similar for two named types:

$$\mathit{Decltype}(e) \simeq \mathit{Decltype}(\varepsilon) \quad \text{or} \quad \mathit{Type_expr}(e) \simeq \mathit{Type_expr}(\varepsilon) \iff e \cong \varepsilon.$$

Syntactic equivalence of expressions are discussed in §6.7.1.

6.5 Product and sum type unification

Two product types or two sum types are unified if and only if their respective sequences (out of which they are constructed) are of the same size, and elements of matching positions are unified.

$$\mathit{Product}(s) \simeq \mathit{Product}(\sigma) \quad \text{or} \quad \mathit{Sum}(s) \simeq \mathit{Sum}(\sigma) \iff \left\{ \begin{array}{l} \mathit{size}(s) = \mathit{size}(\sigma) \\ s_i \simeq \sigma_i, \quad \forall i \end{array} \right\}.$$

6.6 Template type unification

Two template types are unified if and only if they have matching template parameter-lists and the respective parameterized types are unified:

$$\mathit{Template}(p, t) \simeq \mathit{Template}(\pi, \tau) \iff \left\{ \begin{array}{l} \mathit{size}(p) = \mathit{size}(\pi) \\ p_i \cong \pi_i, \quad \forall i \\ t \simeq \tau \end{array} \right\}$$

The syntactic equality of declarations is discussed in §6.7.1.

6.7 From unified types to equivalence of expressions

Various nodes such as those that represent built-in types, typedefs, user-defined type names, dependent (nested-)types, class template specializations are built out of expressions and the type constructors *Type_expr* or *Decltype*. Their unifications depend on the syntactic equivalence of expressions. IPR expression nodes represent either a classic expressions, or a type, or a statement or a declaration. For completeness, we currently unify expressions; but we do not expect that to prove worthwhile. The next subsections give the rules for deciding when two (generalized) expressions are syntactically equivalent.

6.7.1 Classic expressions

Classic expressions are either unary, binary or ternary. Two such expressions are syntactically equivalent if and only if theirs parts are:

$$\begin{aligned} \mathcal{U}(e) \cong \mathcal{U}(\varepsilon) &\iff e \cong \varepsilon, \\ \mathcal{B}(e_1, e_2) \cong \mathcal{B}(\varepsilon_1, \varepsilon_2) &\iff \left\{ \begin{array}{l} e_1 \cong \varepsilon_1 \\ e_2 \cong \varepsilon_2 \end{array} \right\}, \\ \mathcal{T}(e_1, e_2, e_3) \cong \mathcal{T}(\varepsilon_1, \varepsilon_2, \varepsilon_3) &\iff \left\{ \begin{array}{l} e_1 \cong \varepsilon_1 \\ e_2 \cong \varepsilon_2 \\ e_3 \cong \varepsilon_3 \end{array} \right\}, \end{aligned}$$

where \mathcal{U} , \mathcal{B} and \mathcal{T} range over unary, binary and ternary expression constructors, respectively.

6.7.2 Types

Syntactic equivalence of type expressions is type unification as discussed in sections §6.1 through §6.6.

6.7.3 Statements

In Standard C++, and in the current design of IPR, there is no way a statement can appear as argument for a type constructor. However, for completeness, we say that two statement nodes are syntactically equivalent if and only if they are equal (as nodes). As a consequence, a statement node is syntactically equivalent only to itself.

6.7.4 Declarations

Two declarations are syntactically equivalent if and only if

- they have syntactically equal names; and
- they have unified types; and
- they have equal scopes.

Notice that we do not unify declarations, as that is not desirable.

7 Related and Future Work

The IPR library is a direct successor of the *eXtented Type Information* library [13]. XTI was focused on the representation of the C++ type system, whereas IPR aims at the full C++ language. There had been various projects targeting static analysis and transformations of C++ programs. Only a few are active as of today. For example, CodeBoost [1, 2, 7] focuses on transformations of C++ programs, for numerical PDE solvers, written in the Sophus style. It does not handle full Standard C++, as its design criteria was to implement only what is needed in the Sophus project. Simplicisimus [9, 10] and ROSE [8] are other projects for transforming C++ programs.

In the near future, we plan to complete *The Pivot* infrastructure and engineer it to cope with multi-100,000 line real-world programs. This will involve designing a system for representing header files directly (based on unification) and merging separate translation units for multi-translation unit analysis. As a practical test, we will implement conventional style analyzers, statistics gathering, and visualization tools. In the slightly longer term, we will experiment with the use of concepts and library-specific validations, optimizations, and transformations in the domains of parallel, distributed, and embedded systems.

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 programming techniques for efficiency. We use unification to help maintain consistency and to keep our program representation compact, as required for scalability. 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.

References

- [1] 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.
- [2] O. Bagge, K. Kalleberg, M. Haveraaen, and E. Visser, *Design of the CodeBoost transformation system for domain-specific optimisation of C++ programs*, Third International Workshop on Source Code Analysis and Manipulation (SCAM 2003) (Amsterdam, The Netherlands) (Dave Binkley and Paolo Tonella, eds.), IEEE Computer Society Press, September 2003, pp. 65–75.
- [3] International Organization for Standards, *International Standard ISO/IEC 14882. Programming Languages — C++*, 1998.
- [4] International Organization for Standards, *International Standard ISO/IEC 14882. Programming Languages — C++*, 2nd ed., 2003.
- [5] International Organization for Standards, *ISO/IEC PDTR 18015. Technical Report on C++ Performance*, 2003.
- [6] 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>, Septembre 2004, ISO/IEC JTC1/SC22/WG21 no. 1705.
- [7] 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.
- [8] M. Schordan and D. Quinlan, *A Source-to-Source Architecture for User-Defined Optimizations*, 2003, Joint Modular Languages Conference.
- [9] S. Schupp, D. Gregor, D. Musser, and S.-M. Liu, *User-extensible simplification – type-based optimizer generators*, International Conference on

Compiler Construction (R. Wihlem, ed.), Lecture Notes in Computer Science, 2001.

- [10] _____, *Semantic and behavioral library transformations*, Information and Software Technology **44** (2002), no. 13, 797–810.
- [11] B. Stroustrup, *The C++ Programming Language*, special ed., Addison-Wesley, 2000.
- [12] _____, *Concept checking - A more abstract complement to type checking*, <http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2003/n1510.pdf>, Septembre 2003, ISO/IEC JTC1/SC22/WG21 no. 1510.
- [13] _____, *XTI: eXtented Type Information*, Tech. report, AT&T Research, unpublished, 2002.
- [14] B. Stroustrup and G. Dos Reis, *Concepts – Design choices for template argument checking*, <http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2003/n1522.pdf>, Septembre 2003, ISO/IEC JTC1/SC22/WG21 no. 1522.
- [15] _____, *Concepts - syntax and composition*, <http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2003/n1536.pdf>, Septembre 2003, ISO/IEC JTC1/SC22/WG21 no. 1536.