Internal Program Representation for *The Pivot*

G. Dos Reis   B. Stroustrup

1 Introduction

The Pivot framework is a general infrastructure for transformation and static analysis of programs written in C++ programming language. This document serves as a refenrece manual for the internal program representation use in the *The Pivot* framework. A companion document [1] provides a user point of view.

This document is structured as follows. The next subsections introduce general notions of the language used to represent C++ entities. Then §2 go on describing the interface of the various categories of nodes used in IPR. After that, §3 discusses the implementations of those interfaces. Finally, §4 gives a formal definition of XPR, the concrete syntax for the external representation of programs in *The Pivot*.

1.1 The problem, solutions space and design choices

See [7, 2].

1.2 General notions

A program is internally represented as a graph. The nodes represents various notions, such as types, expresssions, declarations, scopes, names...

A (well-formed) program written in C++ is composed of instantiation units. Instantiation units are obtained from translation units after template instantiation requests satisfaction. Consequently, most IPR notions are driven by declarations.

1.2.1 Declaration

A *declaration* is a specification of a *type*, a set of *attributes*, and an optional initialier for a name. The type dictates the major mode of interpretation of the name. The attributes or *declaration specifiers* supply additional interpretations or modifiers the major mode. The initializer is the expression the value of which the name evaluates to, until eventual assignment to another value (if permitted by the type). This notion of declaration covers a wide range of situations. Example:

C++

```
int counter = 0;

struct point {
  int x;
  int y;
};

int next(int x) {
  return x + 1;
}

template<int N>
struct buffer {
   char data[N];
};
```

XPR

```
counter : int = 0;

point : class = {
  x : int;
  y : int;
};

next : (x : int) int = {
  return x + 1;
}

buffer : <N : int> class = {
  data : [N] char;
};
```

In this example, the same program is expressed in both standard C++ and XPR notations. All the declarations follow the same pattern (which is probably more stricking in the XPR notation): a name, a type, and an initializer. In particular, the class body and the function body are the initializers for the class `point` and `next`, repsectively. The name `point` is declared to have type `class`.

In some regions, a declaration can be repeated many times. The first declaration for a name (with a given type) in a given region is called the *master declaration* of that name in that region.

1.2.2 Type

In IPR, a *type* is conceived of as a collection of values or objects with operations or functions. This is not the most abstract definition of type in circulation, but it is sufficiently general and close to the general aims of the *The Pivot* framework. A type in IPR is a notion similar to that of category [6], it is a notion used to describe a collection of objects with given

properties. In similar ways, the notion of category can be used to talk about collection of categories, types in IPR are expressions that have types. For example, the class `ios` of the previous section has type `class`, just like the enumerator `ios::dec` has type `ios::flags` and the enumeration `ios::flags` has type `enum`.

1.2.3 Overloading

Most regions can contain different declarations with a given name. Such declarations are either redeclarations or declarations that use different types. A name is said to be *overloaded* in a given region if that region contains at least two declarations specifying different types for that name. This concept of overloading is uniformly applied in IPR. In particular, any declaration can be overloaded, not just function declaration. So the following fragment (in XPR notation) is valid.

```
pi : double = 3.141592653589793;
pi : float = 3.141592654f;
```

That program frgament cannot be written directly in C++, as only functions can be overloaded in Standard C++. However it could be emulated, in some specific cases, as a function object:

```
struct pi {
  operator double() const { return 3.141592653589793; }
  operator float() const { return 3.141592654f; }
};
```

Given a name in some scope, the collection of all declarations for that name is called its *overload set*.

1.2.4 Scope

A *scope* is a sequence of declarations. This notion is much more general than that of Standard C++. Obviously, the set of all declarations in a given region is a scope. So is the union of the results of looking a name in a collection of regions. That notion sets an elegant way to account for the base classes in a class definition or the enumerators in an enumeration definition. Consider the following declarations:

C++

```
struct ios {
  enum flags {
    boolapha, dec, fixed,
    // ...
  };
  // ...
};

struct istream : virtual ios {
  // ...
};

struct ostream : virtual ios {
  // ...
};

struct stream : istream,
                ostream {
  // ...
};
```

XPR

```
ios :class {
  flags :enum {
    boolapha, dec, fixed,
    // ...
  };
  // ...
};

istream :class (ios public virtual) {
  // ...
};

ostream :class (ios public virtual) {
  // ...
};

stream :class (istream public,
               ostream public) {
  // ...
};
```

The sequence of declarations for `ios`, `istream`, `ostream` and `stream`, appearing at toplevel, forms a scope. So is the sequence of enumerators `ios::booalpha`, `ios::dec`, `ios::fixed`, ... since they are for *emplicitly declared* named constants. Similarly, the sequence of the base classes `istream` and `ostream` of class `stream` is a scope. It is the sequence of *implicitly declared* subjects with the `public` declaration specifier. Finally, the declaration for classes `istream` and `ostream` says that their base scopes consist in a single (implicit) declaration with specifiers `public` and `virtual`.

1.3 Organization

The IPR library is organized into three header files:

1. The interface header file `<ipr/interface>`. This header defines the abstract classes that serve as interface to the implementation classes.

2. The implementation header file `<ipr/impl>`. Implementations for abstract class in the interface header are found there. Their names are derived from the interface my appending the suffix `_impl`.

3. The input and ouput header file `<ipr/io>`. This header provides

functions for reading in memory IPR nodes externally represented in XPR syntax, and writing out IPR nodes in the XPR syntax.

2 Interface Classes

All classes describing the interface of IPR are *immutable* in the sense that they support only `const` operations. There is no way programs using only the interface classes could alter the nodes.

A class of IPR nodes is said *unified* if two nodes of that class have equal values if and only if they have the same identity (as nodes). Some nodes naturally lead themselves to unification in the sense that they are completely determined by their types and the argument list used to create them. Examples of such nodes are those that represent pointer-types, reference-types or identifiers. Other nodes, such as sequences, are first created then "filled up". Unification for such nodes involves some degree of structural equality.

As stated in the introductory section of this paper, most of the IPR notions are driven by the representation of declarations, with full type information. Declarations involve types and expressions (as initializers). C++ expressions share similarity with C++ types in the sense that most of them are obtained as the result of sub-components compositions through unary, binary or ternary operators. That observation is at the basis of the decision of viewing most C++ entities as *generalized expressions*, or simply expressions when the context is clear. In particular, types, statements and declarations are generalized expressions.

2.1 Structural interfaces

As observed above, there is a high degree of common structures between expressions, types and statements. Those commonality are captured in form of unary, binary or ternary expressions, and rendered as the following interfaceo templates:

```
      // -- Unary<> --
template<class Cat = Expr, class Operand = Expr>
struct Unary : Cat {
   typedef Operand Arg_type;
   virtual const Operand& operand() const = 0;
};

      // -- Binary<> --
template<class Cat = Expr, class First = Expr, class Second = Expr>
```

```
struct Binary : Cat {
   typedef First Arg1_type;
   typedef Second Arg2_type;
   virtual const First& first() const = 0;
   virtual const Second& second() const = 0;
};

      // -- Ternary<> --
template<class Cat = Expr, class First = Expr,
         class Second = Expr, class Third = Expr>
struct Ternary : Cat {
   typedef First Arg1_type;
   typedef Second Arg2_type;
   typedef Third Arg3_type;
   virtual const First& first() const = 0;
   virtual const Second& second() const = 0;
   virtual const Third& third() const = 0;
};
```

The first template parameter `Cat` indicates the kind of major categories (e.g. plain nodes, expressions, statements or declarations) the node is a member of. The remaning template parameters indicate the type if subcomponents. The actual node interfaces will define forwarding functions (with more meaningful names) to the generic selectors.

2.2 Universal base class for nodes

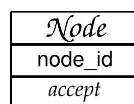

Figure 1: The 𝔑𝔬𝔡𝔢 interface

C++ programs are represented in IPR as graphs, the nodes of which denote (generalizations of) C++ constructs. Any notion directly represented in IPR is so as an object of class derived (either directly or indirectly) from class `Node`. An informal definition of a `Node` is anything of potential interest while traversing graphs of program fragments (through the Visitor Design Pattern [3]). This class is the root of the IPR class hierarchies.

```
struct Node {
    const int node_id;
    Node();
   virtual void accept(Visitor&) const = 0;
};
```

Every node in the IPR representation has a unique identifier (an integer value) assigned to it at its creation. That identifier value is given by the member `Node::node_id`. An alternative would have been to use the node's address as its unique identifier, but using an identifier independent of the address space provides good support for persistency. It also means that we can store nodes in associative containers and use their identifiers as keys, at a relatively efficient and portable fashion.

2.3 Comments in programs

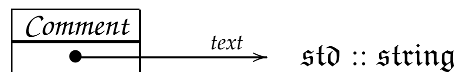

Figure 2: The Comment interface

A comment in C++ programs (be they written in C- or BCPL-style) are represented as unary node, the operand of which is the character stream that makes the comment:

```
struct Comment : Unary<Node, std::string> {
   const std::string& text() const { return operand(); }
};
```

2.4 Annotations

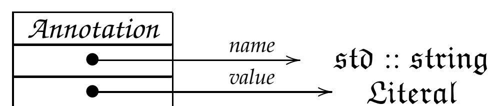

Figure 3: Annotation interface

The external and internal representations of C++ programs allow for annotations on statements. Annotations mostly represent informations not understood by IPR. They are pairs of identifier and literal.

```
struct Annotation : Binary<Node, std::string, Literal> {
   const std::string& name() const { return first(); }
   const Literal& value() const    { return second(); }
};
```

The exact interpretation of an annotation is dependent on the tool that uses the IPR library. Only statements (and therefore declarations) can be directly annotated.

2.5 Regions of program text

Declarations appear inside regions of program texts. Regions contribute to determine validity of name use, objects lifetime, and lexical characterizations of program fragments. Regions nest. Consequently, every region but the global has a parent region.

```
struct Region : Node {
   typedef std::pair<Unit_location, Unit_location> Location_span;
   virtual const Location_span& span() const = 0;
   virtual const Region& enclosing() const = 0;
   virtual const Scope& bindings() const = 0;
   virtual const Expr& owner() const = 0;
};
```

Regions can be delimitated so that they correspond to lexical extent of function or class definitions. A region has an owner, *i.e.* C++ entities they are attached to. Finally, all declarations appearing in a given region form a scope, available through *Region::bindings*.

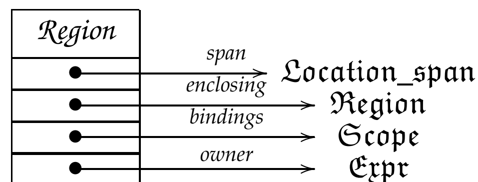

Figure 4: Region interface

2.6 Program unit

A unit is the sequence of toplevel declarations in a transltaion or instantiation unit.

```
struct Unit : Node {
   virtual const Namespace& global_namespace() const = 0;
   virtual const Type& Void() const = 0;
   virtual const Type& Bool() const = 0;
   virtual const Type& Char() const = 0;
   virtual const Type& Signed_char() const = 0;
   virtual const Type& Unsigned_char() const = 0;
   virtual const Type& Wchar_t() const = 0;
   virtual const Type& Short() const = 0;
   virtual const Type& Unsigned_short() const = 0;
   virtual const Type& Int() const = 0;
   virtual const Type& Unsigned_int() const = 0;
   virtual const Type& Long() const = 0;
```

```
    virtual const Type& Unsigned_long() const = 0;
    virtual const Type& Long_long() const = 0;
    virtual const Type& Unsigned_long_long() const = 0;
    virtual const Type& Float() const = 0;
    virtual const Type& Double() const = 0;
    virtual const Type& Long_double() const = 0;
    virtual const Expr& null_expr() const = 0;
};
```

It provides access to C++ fundamental types.

2.7 Generalized expressions

As explained earlier, most C++ notions are represented in IPR as generalized expressions. All IPR expressions are typed. For instance, a declatarion

```
struct Expr : Node {
    virtual const Type& type() const = 0;
};
```

obviously has a type (which is the type specified in the declaration). In particular, a template declaration has a type (as opposed to Standard C++). A type in turn has a type, as explained previously.

2.8 Overload sets

The result of looking up the totallity of declarations for a name in a given scope is an overload set. An overload set is further partitioned into subsets of declarations or redeclarations with same type.

```
struct Overload : Sequence<Decl>, Expr {
    using Sequence<Decl>::operator[];
    virtual const Sequence<Decl>& operator[](const Type&) const = 0;
};
```

An overload set is an expression, therefore has a type. The type of an overload set is a Product type, that accounts for the types of all master declarations.

Selecting a particular member of an overload set, based on its type is supported through the subscription-by-type operation. The result of that operation is the master declaration with that type.

2.9 Interface for scopes

There is only one interface class, `Scope`, that represents the variety of scopes available in a C++ programs (function scope, function prototype scope, local scope, class scope, namespace scope).

The following are general operations supported by all scope nodes.

`const Type& owner(const Scope scope&)` Returns a `const` reference to the `Type` that defines `scope`. Any scope is associated with a given type for which it represents the sequence of associated members or values.

`Scope::operator[](const Name& name) const` Returns a `const` reference to an `Overload` that contains all the master declarations of `name` (in the calling `Scope` object). This set has null size if no declaration is found. The name look up considered here is the *lexical* name lookup.

`Scope`s are visitable, they have an `accept()` member function that takes a reference to any vistor whose type derives from

2.10 Naming entities

2.10.1 `Identifier`

An *identifier* in IPR is anything that Standard C++ defines as identifier, although the IPR library does not make any special enforcement. The string that serves to construct an Identifier is given by the operation *string()*:

Identifiers are expressions, and do belong to a given category. That category is the value of the operation *type()*. Since interpretations of identifiers depend on the context of use, the type of an identifier is a Decltype whose operand is the identifier in question. The IPR interface to the cate-

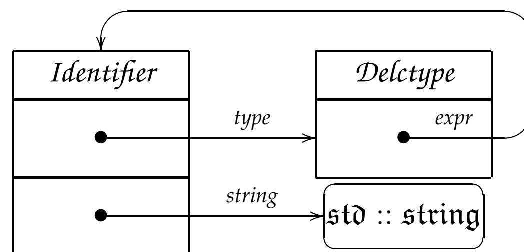

Figure 5: Identifier interface

gory Identifier of identifiers is implemented by the class `Identifier`:

```
struct Identifier : Unary<Name, std::string> {
   const std::string& string() const { return operand(); }
};
```

2.10.2 `Operator`

An IPR *operator name* is any Standard C++ operator name of the form `operator @`, where `@` is a member of OperatorName (see table ???). Similar to an Identifier, the string that served to construct an Operator is given by the operation *opname*:

Also, similar to an Identifier, an Operator supports the *type* operation. The value of that operation is a Decltype whose operand is the Operator.

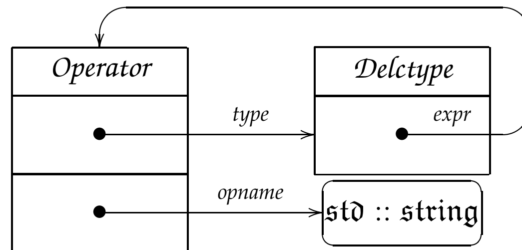

Figure 6: Operator interface

The IPR interface to the operator name category Operator is implemented by the class `Operator`:

```
struct Operator : Unary<Name, std::string> {
   const std::string& opname() const { return operand(); }
};
```

2.10.3 `Conversion`

A *conversion* name is any name made out of `operator` and a type. Given the following holds:

The IPR interface to Conversion is

```
struct Conversion : Unary<Name, Type> {
   const Type& target() const { return operand(); }
};
```

`Conversion::target() const` The type this conversion function converts to.

2.11 Expressions

Nearly everyting is an expression in IPR. An *expression* is understood as the specification of an operator and arguments for a computation. The operator need not be alsways explicitly spelled out.

The various categories of expressions are described in detail in subsequent sections. The interface to this notion is given by

```
struct Expr : Node {
   virtual const Type& type() const = 0;
};
```

Thus, every expression node supports the `type()` and `annotation()` operations:

`Expr::type() const` The IPR node that represent the type of this expression.

`Expr::annotation() const` The sequence of annotations made to this expression.

2.12 Literals

```
struct Literal : Binary<Expr, Type, std::string> {
   const Type& type() const { return first(); }
   const std::string& string() const { return second(); }
};
```

`Literal::string()`

2.13 Types in IPR

The variety of C++ types are represented in IPR. The abstract class `Type` serves as a base class of that hierarchy. A type in IPR is a generalized expression, and therefore derives from the abstract class `Expr`.

2.13.1 Reference type

```
struct Reference : Unary<Type, Type> {
  const Type& refers_to() const { return operand(); }
};
```

2.13.2 Function type

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

Function::signature() const

Function::target() const

Function::throws() const

2.13.3 Pointer type

```
struct Pointer : Unary<Type, Type> {
   const Type& points_to() const { return operand(); }
};
```

Pointer::points_to() const

2.13.4 Array type

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

Array::element_type() const

Array::bound() const

2.13.5 User-defined types

```
struct Udt : Type {
   virtual const Scope& scope() const = 0;
};

struct Namespace : Udt {
   const Sequence<Decl>& members() const { return scope().members(); }
};

struct Class : Udt {
   const Sequence<Decl>& members() const { return scope().members(); }
   virtual const Sequence<Base_type>& bases() const = 0;
   virtual const Scope& base_scope() const = 0;
```

```
};

struct Union : Udt {
   const Sequence<Decl>& members() const { return scope().members(); }
};

struct Enum : Udt {
   virtual const Sequence<Enumerator>& members() const = 0;
};
```

2.14 Unary expressions

2.14.1 Incrementation

2.14.2 Decrementation

2.14.3 Conversions

2.14.4 Type identification

A type identification comes in two flavors

2.14.5 Size of object representation

2.14.6 Dereferencing

2.14.7 Taking the address of an object

2.14.8 Unary plus

2.14.9 Negation

2.14.10 Complement

2.14.11 Deletion

2.15 Binary expressions

2.15.1 Multiplicative expressions

2.15.2 Additive expressions

2.15.3 Logical expressions

2.15.4 Bits arithmetic

2.15.5 Member selection

2.15.6 Expression sequencing

2.15.7 Assignment

2.16 Expressions with variable number of arguments

2.16.1 Function call

2.16.2 Data construction

2.17 Statements

2.17.1 Labeled statements

2.17.2 Expression statements

```
struct Expr_stmt : Stmt {
   virtual const Expr& expr() const = 0;
};

struct Labeled_stmt : Stmt {
   virtual const Expr& label() const = 0;
   virtual const Expr& stmt() const = 0;
};
```

2.17.3 Blocks

```
struct Block : Stmt {
   virtual const Scope& members() const = 0;
   virtual const Sequence<Expr>& stmts() const = 0;
   virtual const Sequence<Handler>& handlers() const = 0;
};
```

2.17.4 Ctor bodies

```
struct Ctor_body : Stmt {
   virtual const Expr_list& inits() const = 0;
   virtual const Block& block() const = 0;
};
```

2.17.5 Selection statements

```
struct If_then : Stmt {
   virtual const Expr& condition() const = 0;
   virtual const Expr& then_stmt() const = 0;
};

struct If_then_else : Stmt {
   virtual const Expr& condition() const = 0;
   virtual const Expr& then_stmt() const = 0;
   virtual const Expr& else_stmt() const = 0;
};

struct Switch : Stmt
   virtual const Expr& condition() const = 0;
   virtual const Expr& body() const = 0;
;
```

2.17.6 Iteration iterations

```
struct While : Stmt {
   virtual const Expr& condition() const = 0;
   virtual const Expr& body() const = 0;
};

struct Do : Stmt {
   virtual const Expr& condition() const = 0;
   virtual const Expr& body() const = 0;
};

struct For : Stmt {
   virtual const Expr& initializer() const = 0;
   virtual const Expr& condition() const = 0;
```

```
      virtual const Expr& increment() const = 0;
      virtual const Expr& body() const = 0;
   };
```

2.17.7 Jump statements

```
   struct Break : Stmt {
      virtual const Expr& from() const = 0;
   };

   struct Continue : Stmt {
      virtual const Expr& iteration() const = 0;
   };

   struct Goto: Stmt {
      virtual const Expr& target() const = 0;
   };

   struct Return : Stmt {
      virtual const Expr& value() const = 0;
      const Type& type() const   return value().type();
   };
```

2.17.8 Exception handlers

```
   struct Handler : Stmt {
      virtual const Decl& exception() const = 0;
      virtual const Block& body() const = 0;
   };
```

2.18 Declarations

```
   struct Decl : Stmt {
      enum Specifier {
         None = 0,
         Auto        = 1 << 0,
         Register    = 1 << 1,
         Static      = 1 << 2,
         Extern      = 1 << 3,
         Mutable     = 1 << 4,
         StorageClass  = Auto | Register | Static | Extern | Mutable,

         Inline      = 1 << 5,
         Virtual     = 1 << 6,
         Explicit    = 1 << 7,
         Pure        = 1 << 8,
         FunctionSpecifier = Inline | Virtual | Explicit | Pure,
```

```
        Friend     = 1 << 9,
        Typedef    = 1 << 10,

        Public     = 1 << 11,
        Protected  = 1 << 12,
        Private    = 1 << 13,
        AccessProtection = Public | Protected | Private
    };

    virtual Specifier specifiers() const = 0;
    virtual const Name& name() const = 0;
    virtual const Scope& scope() const = 0;

    virtual bool has_initializer() const = 0;
    virtual const Expr& initializer() const = 0;

    const Sequence<Decl>& decl_set() const
    { return scope()[name()][type()]; }
    const Decl& master() const { return *decl_set().begin(); }
};
```

A declaration is a statement. It usually introduces a name in a given scope and attributes for its interpretation: (a) a type; (b) a set of specifiers for access or storage; and (b) an optional initializer.

2.18.1 Declaration specifiers

Storage class specifiers

Function specifiers

Access control and friendship

2.18.2 Variable declarations

```
struct Var : Decl {
};
```

2.18.3 Data member declarations

```
struct Field : Decl {
};

struct Bit_field : Decl {
```

```
   virtual const Expr& size() const = 0;
};
```

2.18.4 Parameter declarations

```
struct Parameter : Decl {
   virtual int position() const = 0;
   const Expr& default_value() const  return initializer();
};
```

2.18.5 Function declarations

```
struct Fun_decl : Decl {
   virtual const Type& target() const = 0;
   virtual const Parameter_list& parameters() const = 0;
   virtual const Type& throws() const = 0;
};
```

Constructors

```
struct Constructor : Fun_decl {
   const Name& name() const { return membership().name(); }
   const Type& target() const { return membership(); }

   virtual const Type& membership() const = 0;
};
```

Destructors

```
struct Destructor : Fun_decl {
   virtual const Type& membership() const = 0;
};
```

2.18.6 Alias declarations

```
struct Alias : Decl {
};
```

2.18.7 Base-type declarations

```
struct Base_type : Decl {
   const Name& name() const { return type().name(); }
};
```

2.18.8 Enumerator declarations

```
struct Enumerator : Decl {
   const Type& type() const { return membership(); }
   virtual const Enum& membership() const = 0;
};
```

2.18.9 Type declarations

```
struct Type_decl : Decl {
};
```

2.19 Traversing graphs: Visitors

```
struct Visitor {
   virtual void visit(const Node&) = 0;
   virtual void visit(const Annotation&);

   virtual void visit(const Expr&) = 0;

   virtual void visit(const Name&) = 0;
   virtual void visit(const Identifier&);
   virtual void visit(const Operator&);
   virtual void visit(const Conversion&);
   virtual void visit(const Type_id&);

   virtual void visit(const Type&) = 0;
   virtual void visit(const Array&);
   virtual void visit(const Class&);
   virtual void visit(const Enum&);
   virtual void visit(const Expr_as_type&);
   virtual void visit(const Function&);
   virtual void visit(const Namespace&);
   virtual void visit(const Pointer&);
   virtual void visit(const Product&);
   virtual void visit(const Reference&);
   virtual void visit(const Template&);
   virtual void visit(const Union&);
   virtual void visit(const Udt&);

   virtual void visit(const Expr_list&);
   virtual void visit(const Overload&);
   virtual void visit(const Scope&);

   virtual void visit(const Address&);
   virtual void visit(const Array_delete&);
   virtual void visit(const Complement&);
   virtual void visit(const Delete&);
```

```
virtual void visit(const Deref&);
virtual void visit(const Expr_sizeof&);
virtual void visit(const Expr_stmt&);
virtual void visit(const Expr_typeid&);
virtual void visit(const Label&);
virtual void visit(const Negate&);
virtual void visit(const Not&);
virtual void visit(const Post_decrement&);
virtual void visit(const Post_increment&);
virtual void visit(const Pre_decrement&);
virtual void visit(const Pre_increment&);
virtual void visit(const Throw&);
virtual void visit(const Type_sizeof&);
virtual void visit(const Type_typeid&);
virtual void visit(const Unary_plus&);

virtual void visit(const Add&);
virtual void visit(const Add_assign&);
virtual void visit(const And&);
virtual void visit(const Array_ref&);
virtual void visit(const Arrow_select&);
virtual void visit(const Arrow_star&);
virtual void visit(const Assign&);
virtual void visit(const Bitand&);
virtual void visit(const Bitand_assign&);
virtual void visit(const Bitor&);
virtual void visit(const Bitor_assign&);
virtual void visit(const Bitxor&);
virtual void visit(const Bitxor_assign&);
virtual void visit(const C_cast&);
virtual void visit(const Call&);
virtual void visit(const Comma&);
virtual void visit(const Const_cast&);
virtual void visit(const Div&);
virtual void visit(const Div_assign&);
virtual void visit(const Dot_select&);
virtual void visit(const Dot_star&);
virtual void visit(const Dynamic_cast&);
virtual void visit(const Equal&);
virtual void visit(const Greater&);
virtual void visit(const Greater_equal&);
virtual void visit(const Less&);
virtual void visit(const Less_equal&);
virtual void visit(const Literal&);
virtual void visit(const Member_init&);
virtual void visit(const Modulo&);
virtual void visit(const Modulo_assign&);
virtual void visit(const Mul&);
virtual void visit(const Mul_assign&);
```

```
    virtual void visit(const Not_equal&);
    virtual void visit(const Object_construction&);
    virtual void visit(const Or&);
    virtual void visit(const Reinterpret_cast&);
    virtual void visit(const Scope_ref&);
    virtual void visit(const Shift_left&);
    virtual void visit(const Shift_left_assign&);
    virtual void visit(const Shift_right&);
    virtual void visit(const Shift_right_assign&);
    virtual void visit(const Specialization&);
    virtual void visit(const Static_cast&);
    virtual void visit(const Sub&);
    virtual void visit(const Sub_assign&);

    virtual void visit(const Conditional&);
    virtual void visit(const New&);

    virtual void visit(const Stmt&) = 0;
    virtual void visit(const Labeled_stmt&);
    virtual void visit(const Block&);
    virtual void visit(const Ctor_body&);
    virtual void visit(const If_then&);
    virtual void visit(const If_then_else&);
    virtual void visit(const Switch&);
    virtual void visit(const While&);
    virtual void visit(const Do&);
    virtual void visit(const For&);
    virtual void visit(const Break&);
    virtual void visit(const Continue&);
    virtual void visit(const Goto&);
    virtual void visit(const Return&);
    virtual void visit(const Handler&);

    virtual void visit(const Decl&) = 0;
    virtual void visit(const Alias&);
    virtual void visit(const Base_type&);
    virtual void visit(const Parameter&);
    virtual void visit(const Parameter_list&);
    virtual void visit(const Var&);
    virtual void visit(const Field&);
    virtual void visit(const Bit_field&);
    virtual void visit(const Fun_decl&);
    virtual void visit(const Constructor&);
    virtual void visit(const Destructor&);
    virtual void visit(const Type_decl&);
};
```

3 Implementation Classes

Every abstract class in the interface class hierarchy has a corresponding implementation class, named after that interface and with the suffix `Impl`. Those implementations classes often supports non-`const` operations too. Every implementation class provide an overriding implementation for the visitor hook `Node::accept()` that transfer control to the appropriate overriding member function of `Visitor::visit()`

There is only one instance of `std::string` for any given text string that might be hold in any IPR node (e.g. by names).

3.1 Implementation classes for names

The four kinds of names *identifier*, *operator name*, *conversion-function name* and *template instantiation name* are respectively implemented by `impl::Identifer`, `impl::Operator`, `impl::Conversion` and `impl::Specialization`. They all derive from instantiations of the parameterized implementation class `impl::Name<>`. The later is parameterized by the interface

3.1.1 `impl::Identifier`

3.1.2 `impl::Operator`

3.1.3 `impl::Conversion`

4 XPR grammars

XPR essentially retains C++'s syntax for expressions. The differences appear mainly in the declaration syntax where XPR aims for a linear notation.

4.1 Lexical conventions

4.1.1 Literals

XPR retains C [4] and C++ [5]'s spelling for literals

References

[1] G. Dos Reis, *IPR User Guide*, 2004.

[2] G. Dos Reis and B. Stroustrup, *Representing C++ Directly, Efficiently and Completely*, 2005.

[3] E. Gamma and al., *Design Patterns*, Addison-Wesley, 1994.

[4] ISO, *International Standard ISO/IEC 9899. Programming Languages — C*, 2nd ed., 1999.

[5] ISO, *International Standard ISO/IEC 14882. Programming Languages — C++*, 2nd ed., 2003.

[6] S. Mac Lane, *Categories for the Working Mathematicians*, 2nd ed., Springer, 2001.

[7] B. Stroustrup, *XTI: A simple, complete C++ Extended Type Information Library*, 2002.