IPR User Guide

(for version 0.34)

March 23, 2006

1 Introduction

This document is a companion to the IPR Reference Manual [?]. It is intended to illustrate uses of the IPR library.

The library currently consists of three components:

- *the interface*, available through `<ipr/interface.H>`. This is a collection of interfaces (*i.e.* abstract classes) that provide views and non-mutating operations to manipulate IPR nodes;
- *the I/O component*, available through `<ipr/io.H>`. This header file declares functions necessary to render IPR nodes in their external, persistent form according to the XPR syntax;
- *the implementation classes*, available through `<ipr/impl.H>`. This is a collection of types that provide implementations for the interface component. They also support mutating operations necessary to build complete IPR nodes.

Programs that use the IPR library usually include `<ipr/interface.H>` when their only interests are non-mutating manipulations of IPR nodes. They need to include `<ipr/io.H>` if they intend to print out IPR nodes in XPR syntax. Finally, they include `<ipr/impl.H>` if they do create IPR nodes, as opposed to inspecting them.

The interface classes reside in the namespace `ipr`. The implementation classes are found in the sub-namespace `ipr::impl`. In general, an interface `ipr::Xyz` has a corresponding implementation named `ipr::impl::Xyz`.

2 Installation

3 Generalities

The IPR library provides data structures for representing programs written in ISO Standard C++. Programs are represented as graphs, with distinguished roots called *units*. An IPR unit roughly corresponds to a translation unit in C++. In fact, IPR represents a superset of Standard C++; so it can handle programs using full C++ as well as incorrect or incomplete programs.

The notion of unit is directly represented by the interface class `Unit`. An object of that type contains the sequence of top-level declarations in a given translation unit. It also provides access to nodes that represent C++ fundamental types. The general approach is that every C++ entity has a type. In particular C++ types, being expressions in IPR, do have types. For example, the following program prints out some IPR expression nodes and their types.

```
#include <ipr/impl.H>
#include <ipr/io.H>
#include <iostream>

template<class E>
void inspect(const E& e)        // Print a expression and its type.
{
   ipr::Printer pp(std::cout);  // Create an XPR pretty printer
                                // tied to std::cout.
   pp << ipr::xpr_expr(e);      // Output e in XPR notation.
   pp << " has type ";
   pp << ipr::xpr_expr(e.type()); // Output type of e in XPR notation.
   std::cout << std::endl;
}

int main()
{
   using namespace ipr;
   impl::Unit unit;             // Current translation unit

   // Inspect the representations of some literals.
   inspect(*unit.make_literal(unit.Char(), "'c'"));
   inspect(*unit.make_literal(unit.Int(), "1024"));
   inspect(*unit.make_literal(unit.Double(), "3.141592653589793"));

   // Inspect the representations of some C++ fundamental types.
   inspect(unit.Char());
   inspect(unit.Int());
   inspect(unit.Double());
}
```

When, compiled and linked against the IPR library, it produces the output

```
'c' has type char
1024 has type int
3.141592653589793 has type double
char has type  typename
int has type  typename
double has type  typename
```

A node of type `Unit` also serves to represent the notion of *instantiation unit*[1]. Recall that typical C++ translation units contain requests for template instantiations. An instantiation unit is a translation unit where all template instantiation requests have been resolved.

The next sections will focuse on the part of the library typically used for building IPR nodes. It is meant to provide support for tools that create IPR nodes, e.g. IPR generators from compilers or XPR parsers. All IPR node construction functions return pointers; usually their names start with the prefix `make_`.

Type nodes, expression nodes and statements nodes are created with `impl::Unit` objects. Declarations are created with their enclosing regions. They are generally represented as a triple of name, type and optional initializer.

4 Literals

Lexical units (e.g. names, literals) of Standard C++ are directly represented in IPR. This section deals with literals and the next with names.

ISO Standard C++ defines five sub-categories of literals:

literal:
    integer-literal
    character-literal
    floating-literal
    string-literal
    boolean-literal

They are all uniformly represented in IPR as pairs of type and string. There is no restriction on the set of types a literal can be conceived for. In other

[1]See clause 2 of [**?**, **?**]

words, any type node in IPR can serve to create a literal. The semantics of the resulting node is entirely up to the environment that creates and, or interprets such nodes.

As the example from §3 shows, literals are made with the member function `make_literal()` of the implementation class `impl::Unit`. It takes two arguments: the type of the literal and the spelling string. There is no mutating operations on literals. The sequence of characters should include back-quotes or double-quotes for character literals or string literals.

5 Names

Names are useful for designating data, types or any computational entity of interest. IPR covers the variety of names found in Standard C++. All names are unified in the sense that, two calls to the same name-constructor function, with the same argument list, will yield the same physical node. Consequently, all name nodes are immutable.

There are eight sub-categories of names

name:
    identifier
    operator-function-id
    conversion-function-id
    qualified-id
    template-id
    type-id
    constructor-name
    destructor-name

accounting for the variety of names in Standard C++.

5.1 Plain identifiers

identifier:
    nondigit
    identifier nondigit
    identifier digit

An `Identifier` node can be created out of a string that contains any character string:

```
unit.make_identifier(string)
```

where *`unit`* is an instance of `impl::Unit` and *`string`* is the string to make an identifier out of. The whole expression as type `const Identifier*`.

5.2 Operator function names

operator-function-id:
    `operator` *operator*

Any operator-name node can be constructed out of a string containing the spelling of that operator:

```
unit.make_operator(string);
```

where *`unit`* is an expression of type `impl::Unit` and *`string`* is a string for the operator-name. The whole expression has type `const Operator*`.

For usual operators like `operator+` or `operator==`, the string is `"+"` or `"=="`. For the array forms of the allocating and deallocating functions, the string is `"new[]"` or `"delete[]"`.

5.3 Conversion function names

conversion-function-id:
    `operator` *type-id*

IPR represents a conversion function name as a node that takes the destination type as argument. Such node is constructed as:

```
unit.make_conversion(type)
```

where *`unit`* is an expression of type `impl::Unit` and *`type`* is an expression of type derived `Type`. The return value is of type `const Conversion*`.

5.4 Qualified names

qualified-id:
    `::`$_{opt}$ *nested-name-specifier unqualified-name*
    `::` *name*

IPR represents qualified names as a `Scope_ref` that takes two arguments: (a) the qualifying expression; and (b) the designated name.

```
unit.make_scope_ref(scope, name)
```

where *`unit`* is of type `impl::Unit`, *`scope`* — designating the qualifying part — is any node of type derived from `Expr`, and *`name`* — designating the name being qualified — is also any node of type derived from `Expr`. The return value is of type `const Scope_ref*`.

5.5 Template specialization names

template-id:
  unqualified-name <template-argument-list$_{opt}$>

Template specialization names, `Specialization` in IPR, are made out of two expressions, as:

```
unit.make_specialization(template, arglist);
```

where the first argument designates the template to specialize and the second argument is an `Expr_list` that denotes the template-argument list. The return value has type `const Specialization*`.

5.6 Type names

All types have names. Type-names may be as simple as `int` or `double`; but they can also be less simpler and not composed not just a single identifier, e.g. `int (*[256])(int)`. Names for such constructed types are represented in IPR by `Type_id` nodes. They can be built as:

```
unit.make_type_id(type)
```

that takes the to-be-named type as its sole argument. Usually, IPR users should not need to have to create a `Type_id` nodes by themselves as these get created automatically for compound types that need them.

5.7 Constructor name

Standard C++ says that constructors don't have names. However, it also says that some qualified names actually designate constructors. Which is quite confusing. For example, in the program fragment

```
struct S {
   // ...
};
S x = S();
```

the declaration of the variable `x`, the symbol `S` refers to two distinct entity: The first is the class-type and the second is the default constructor (invoked through `S()`). Similarly, a class with a template constructor needs ways to refer to partial specializations and explicit instantiations.

IPR defines a category of name called `Ctor_name` used to represent constructor names. A node for such type can be built as

```
unit.make_ctor_name(type)
```

where *`type`* is the type whose constructor one wants to build a name for. The resulting expression has type `const Ctor_name*`.

5.8 Destructor name

The node class `Dtor_name` represents a destructor name. Say

```
unit.make_dtor_name(type)
```

to make a destructor name for *`type`*. The resulting expression has type `const Dtor_name*`.

6 Types

IPR has a unified type system. By that we mean that any two calls to the same type node construction function with the same argument list will yield the same physical node.

6.1 CV-qualified type

Any type (in IPR sense) can be cv-qualified. The set of cv-qualifiers currently supported consists in `const`, `volatile` and `restrict` (C99). They are denoted by the enumerators `Type::Const`, `Type::Volatile` and `Restrict`. They can be bitor-ed. A cv-qualified type is built with the `impl::Unit` member function

```
const Type* make_cv_qualified(const Type&, Type::Qualifier);
```

where the first parameter denotes the type to be cv-qualified and the second parameter denotes the set of cv-qualifier to apply.

6.2 Pointer type

A `Pointer` type node can be constructed out of any IPR type node with the `impl::Unit` member function

```
const Pointer* make_pointer(const Type&);
```

6.3 Reference type

Similar to pointer type nodes, a `Reference` type node can be constructed out of any IPR type node with the `impl::Unit` member function

```
const Reference* make_reference(const Type&);
```

6.4 Array type

An array is made with the `impl::Unit` member function

```
const Array* make_array(const Type&, const Expr&);
```

where the first argument is the array element type and the second argument specifies the bound. For example, ISO Standard C++ says that the type of the string literal `"Hello World!"` is `const char[13]`, so its type will be represented in IPR with a node constructed as

```
unit.make_array(*unit.make_cv_qualified(unit.Char()),
                *unit.make_literal(unit.Int(), "13"));
```

As a limiting case, an array type with no specified bound, e.g. `int[]`, may be constructed along the lines of

```
unit.make_array(unit.Int(), unit.null_expr());
```

where `unit.null_expr()` denotes here the absence of bound specification.

6.5 Declared type

IPR has provision to represent the mechanism to query generalized expression types (see the `auto` and `decltype` proposals). Such an explicit request for inferred type is represented by a `Decltype` node, constructed with

```
const Decltype* make_decltype(const Expr&);
```

6.6 Expression as type

In various contexts it is useful to view an expression node as actually denoting a type. For instance, the C++ reserved identifier `bool` designates the built-in boolean type. The way that is accomplished in IPR is to build an `As_type` node out of the `Identifier` node created for `bool`:

```
unit.make_as_type(*unit.make_identifier("bool"))
```

All built-in C++ simple types are created that way.

Another instance of situations where an `As_type` is helpful is when dealing with dependent types. Assume that you have created a `Parameter` node for a template-parameter `C` and you want to represent the nested-type `typename C::value_type`. The natural answer here is to build a `Scope_ref` and use the resulting node as a type (reflecting the assumption introduced by the C++ keyword `typename`):

```
unit.make_as_type(*unit.make_scope_ref
                      (C, *unit.make_identifier("value_type")))
```

6.7 Function type

A function type consists in three logical parts: (a) the parameter type list; (b) the return type; and (c) the list of exceptions a function of that type may throw. IPR represents a function type as a `Function` node. Such nodes can be constructed with the `impl::Unit` member functions

```
const Function* make_function(const Product&, const Type&);
const Function* make_function(const Product&, const Type&, const Sum&);
```

The first constructor function is a convenient surrogate for the second, as it implies that the function may throw any type (noted `...`).

6.8 Template type

A template type is very similar to a function type, except that it is represented by a `Template` node. The corresponding constructor function is the member function of `impl::Unit`

```
const Template* make_template(const Product&, const Type&);
```

The first argument is the type of the parameter list, and the second argument is the type of the expression being parameterized.

6.9 Pointer to member type

A pointer to member type is represented as a pair: The class-type and the type of the member. The corresponding construction function is

```
const Ptr_to_member* make_ptr_to_member(const ipr::Type&,
                                        const ipr::Type&);
```

of the class `impl::Unit`.

At the implementation level, most compilers distinguish pointer to data member types from pointer to member function types. However, the internal representation is immaterial to the actual semantics constraints. One issue with our desire to have uniform syntax and representation for non-static member and ordinary functions, however, is to make sure that address of members are given the right type: this is easily solved as at anytime we do know when a function declaration is a non-static member function. See the discussion in 9.

6.10 User-defined types

All standard C++ user-defined types (e.g. class or struct, union, enum) as well as namespace are considered user-defined types in IPR. They can be constructed with one of the expressions

```
unit.make_class(parent_region);
unit.make_enum(parent_region);
unit.make_namespace(parent_region);
unit.make_union(parent_region);
```

The resultinf type is `impl::Class*`, `impl::Enum*`, `impl::Namespace*` and `impl::Union*`, respectively.

The definitions of all user-defined types delimit a region, which is the declarative region of their members. All, except enums, have a heterogeneous region, implemented by the class `impl::Region`. Enums' bodies are homogeneous region (`impl::homogeneous_region`) because their members are all enumerators of the same type. For instance, an enumeration's body is a `impl::homogeneous_region<Enumerator>`

Similarly the sequence of declarations in a heterogeneous region is a heterogeneous scope (`impl::Scope`), whereas in a homogeneous region, it is a `impl::homogeneous_scope<T>`, where `T` is the interface type of the declarations.

7 Classic expressions

The abstract language modeled by IPR nodes is expression-based. This is to say that nearly any computational entity of interest is thought of as an expression. Thus compared to Standard C++ notions, most IPR nodes may be considered as representations of *generalized expressions*. Consequently, the term *classic expressions* is used to designate genuine Standard C++ expressions.

Classic expressions are generally divided into categories:

- *unary expressions*, e.g. `sizeof (int)`;
- *binary expressions*, e.g. `a + b`;
- *ternary expressions*, e.g. `x < y ? x : y`.

Each category contains a comprehensive list of IPR nodes specifically designed to cover all kinds of expressions that fall in that category.

Every expression is made by the corresponding `Unit` object. For example, the following program fragment shows how to create an IPR node for `sizeof (int)`:

```
unit.make_type_sizeof(unit.Int());
```

Similarly, here is how one can construct nodes for the expression `a + b`

```
Var* a = ...
Var* b = ...
Expr* sum = unit.make_plus(a, b);
```

The conditional expression `x < y ?  x :  y` can be constructed as follows

```
Expr& x = ...
Expr& y = ...
Expr& min = *unit.make_conditional(*unit.make_less(x, y), x, y);
```

Cast-expressions are represented as binary expressions: The first operand is the target type and the second operand is the source expression.

7.1 User-defined operators

In Standard C++, most operators can be overloaded and given a user-defined meaning. For generality, IPR takes the approach that just about any operator can be given a user-defined meaning. Consequently, every IPR implementation class for a operator has a field `op_impl` to record the function declaration that implements that operator. For instance, consider the expression `cout « 2` for inserting the integer value tow in the output standard stream. Assume further that the insertion function operator declaration is represented by `ins`, then the IPR representation for the whole expression would be constructed as

```
impl::Lshift* shift = unit.make_lshist(cout, two);
shift->op_impl = ins;   // remember the insertion function
```

If the member `op_impl` is not set, then it is assumed to be a built-in operator.

7.2 Names and lookup resolutions

8 Named data

This section describes the representation of named data, *i.e.* variables, data member. Notice that in either cases, references are treated no specially from variables or data members – even though the C++ standard says that references are not objects.

Named data are declared at user-defined types (*i.e.* class-types, enumerations or namespaces) level. Consequently, their IPR node representations are manifactured with a member function, prefixed by `declare_`, of the enclosing user-defined type. The next subsections discuss the mapping ofthe variety C++ named data to IPR representation.

8.1 Variable declaration

A variable is an association of a name and a storage for data, that does not have any enclosing object. The notion of variable is implemented by `impl::Var`. Manifacturing a node of that type requires — as do most declaration nodes — three things: (a) the name of the variable; (b) the type of the data to store; and (c) an optional initializer.

A node for a variable is created with the member function `declare_var()` of the implementation class for the enclosing user-defined type (`impl::Class`, `imp::Union` or `impl::Namespace`). Once a `impl::Var` is created, it must be associated with its initializer (if any). The following program illustrates how to build a variable declaration

```
  static const int bufsz = 1024;
```

at the global scope and write the resulting node in XPR notation.

```
#include <ipr/impl.H>
#include <ipr/io.H>
#include <iostream>
int main()
{
   using namespace ipr;
   impl::Unit unit;              // current translation unit

   // Build the variable's name and type
   const Name* name = unit.make_identifier("bufsz");
   const Type* type = unit.make_cv_qualified(unit.Int(), Type::Const);

   // and the actual impl::Var node at global scope and initialize it
   impl::Var* var = unit.global_ns->declare_var(*name, *type);
   var->decl_data.spec = ipr::Decl::Static;
   var->init = unit.make_literal(unit.Int(), "1024");

   // Print out the whole translation unit
   Printer pp(std::cout);
   pp << unit;
   std::cout << '\n';
}
```

The output of the above program is

```
  bufsz : static const int = 1024;
```

The effort of constructing the representation of that declaration basically is one line per significant token.

8.2 Static data members

A static data member is a global variable in disguise. They key difference between a static data member and an ordinary variable that the former is subject to to access control. Therefore, a static data member is also represented by a `Var` node. From the IPR representation point of view, the steps for manifacturing a node for a static data member is pretty much the same as for ordinary variable, except that one uses the enclosing class-type as the manifacturer. For instance, consider the program fragment

```
  class nifty_counter {
    // ...
  private:
     static int count;
  };
```

where we are only interested in the portion relating to the representation of the static data member `nifty_counter::count`. Here is a program that accomplishes that job and prints the IPR representation:

```
#include <ipr/impl.H>
#include <ipr/io.H>
#include <iostream>

int main()
{
   using namespace ipr;
   impl::Unit unit;              // current translation unit

   // First, build the node for the nifty_counter class
   impl::Class* nifty_type = unit.make_class(*unit.global_region());
   nifty_type->id = unit.make_identifier("nifty_counter");
   unit.global_ns->declare_type(*nifty_type->id, unit.Class())->init
      = nifty_type;

   // Then build the static data member
   impl::Var* count = nifty_type->declare_var(*unit.make_identifier("count"),
                                              unit.Int());
   // "cout" is private
   count->decl_data.spec = Decl::Static | Decl::Private;
   // We do not set the initializer, since there is none.

   // Print out the whole translation unit
   Printer pp(std::cout);
   pp << unit;
   std::cout << '\n';
}
```

A `Var` node represents a static data member if and only if its membership is a class-type. Notice that the operation `declare_var()` is supported by implementation classes for all user-defined types, except enumerations.

8.3 Non-static data members

A non-static data member has semantics different from that of ordinary variable and static data-members. Consequently, it is represented by a distinct node (`Field`). However, the steps for manifacturing such node is very similar to that of a variable, except that the construction function `declare_field()` is used — instead of `declare_var()`.

Consider the following C++ program fragment

```
  struct point {
    int x;
    int y;
  };
```

that declares a class named `point` at the global scope. That declaration can be constructed as follows

```
#include <ipr/impl.H>
#include <ipr/io.H>
#include <iostream>

int main()
{
   using namespace ipr;
   impl::Unit unit;

   // Make the class node, and bind it to the name "point".
   impl::Class* point = unit.make_class(*unit.global_region());
   point->id = unit.make_identifier("point");
   unit.global_ns->declare_type(*point->id, unit.Class())->init = point;

   // declare the public data member "x".
   point->declare_field(*unit.make_identifier("x"), unit.Int())
      ->decl_data.spec = Decl::Public;

   // Ditto for "y".
   point->declare_field(*unit.make_identifier("y"), unit.Int())
      ->decl_data.spec = Decl::Public;

   // Print the current unit.
   Printer pp(std::cout);
   pp << unit;
   std::cout << std::endl;
}
```

The output is:

```
point : class {
    x : public int;
    y : public int;
};
```

In a well formed C++ programs, a `Field` does not have an initializer. However, the IPR library does not enforce that rule — an erroneous program may initialize a non-static data member and we do not gain any uniformity in enforcing that rule in the IPR data structures.

9 Function declarations

This sections discusses the basics for building nodes for function declarations. A function declaration is an association of a name with a mapping. For example, the C++ declaration

```
  int identity(int);
```

defines a mapping from `int` to `int` called `identity`. More generaly, that declaration is represented as the association of the name `identity` with a unary parameterized expression, where the argument is expected to be of type `int`, the result of type `int`, and the computation of the result may be abnormally ended with an exception of any type (implied by the absence of exception specification). In XPR notation, it reads:

```
identity : (:int) int throw (...);
```

Similar to the situation of named data (§8), C++ distinguishes member functions from non-member functions. Unlike the named data case, both member functions and non-member function declarations are uniformly represented by `Fundecl` nodes. Like in the `Var` case, the membership of a `Fundecl` distinguishes a non-member function (membership is a namespace) from a member function (membership is a class-type).

Member functions are further subdivided into two categories: static member functions and non-static member functions. The former are disguised non-member functions with privileged access to their enclosing class-type's members (kind of friend non-member functions), and the latter have an implicit or implied parameter. We do not have different kinds of function types for the types of static member functions and non-static member functions. They all have types uniformely represented by a `Function` node.

A `Fundecl` for a static member function differs from a `Fundecl` for a non-static member function in that the former has the `Decl::Static` set and the latter not.

Furthermore, the correspondance between a non-static member function in C++ and its "regularized" version in IPR makes and adjustment: the keyword `this` is made an explicit parameter. For instance, the following member function declarations

```
struct A
  // ...
  int& f();
  const int& f() const;
  static double g(A*);
  virtual void h(int) = 0;
;
```

will be interpreted as (XPR notation)

```
f: public (this: *A) &int throw(...);
f: public (this: *const A) &const int throw(...);
g: public static (: *A) double throw(...);
h: public virtual pure (this: A*, : int) void throw(...);
```

As a general rule, every non-static member function is adjusted to take `this` as first parameter, of type `*`*`cv T`* where *`T`* is the class and *`cv`* is the cv-qualification of the non-static member function.

9.1 Mapping

A mapping is a parameterized expression. The notion of mapping is general enough to account for both function declaration bodies and template declaration bodies. To illustrate the generality here, consider the following program fragments in both Standard C++ and XPR notations:

C++

```
int identity(int x) {
  return x;
}

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

XPR

```
identity : (x : int) int throw(...) = {
  return x;
}

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

A feature of the XPR notation here is that it makes a clear separation between the names being declared and what they are being bound to. The function `identity` maps an `int` to an `int`, and the class template `buffer` maps an `int` to a class-type. Furthermore, how the integer result of `identity` is computed is given by `int { return x; }`; similarly how the class result of `buffer` is computed is given by `struct { char data[N]; }`. The implementation of either map constitutes its body.

A mapping consists of a parameter list and a body. Its IPR implementation node is described by the class`impl::Mapping`, built with the member function

```
impl::Mapping* make_mapping(const ipr::Region&);
```

of the `impl::Unit` class. For example, let's assume the previous declaration for `identity` appears at the global scope. Then, its associated mapping node would be made as follows:

```
impl::Mapping* mapping = unit.make_mapping(*unit.global_region());
```

Parameters are specified with the `param()` member function of `impl::Mapping`.

```
impl::Parameter* x = mapping->param(*unit.make_identifier("x"), unit.Int());
```

The data member `impl::Mapping::parameters` holds the list of specified parameters. Its type is a product type, and represented by a `Product` node. It describes the domain type of the mapping. A mapping, being an expression, as a type. That type is represented by a `Function` node when it is associated with a function declaration, and a `Template` node when it is associated with a parameterized declaration. So, for the case of the `identity` function declaration, one would write the following

```
const Function* ftype = unit.make_function
    (mapping->parameters.type(), unit.Int());
mapping->constraint = ftype;
```

We will discuss the case of parameterized declarations in §11.

9.2 Naming a mapping

Building a `impl::Fundecl` is very similar to the process of building a node for a variable: one needs a name, a type and optional initializer. As explained above, the initializer for a function declaration is a mapping.

```
impl::Fundecl* f = unit.global_ns->declare_fundecl
   (*unit.make_identifier("identity"), *ftype);
f->init = mapping;                         // the named mapping
```

The node for a function declaration that is not a definition is initialized with an incomplete mapping. An incomplete mapping is a mapping whose body is not specified.

9.3 Constructors and destructors

A constructor or destructor is represented as a non-static member function, suitably adjusted to take `this` as a first parameter. Constructors and destructors to not return values, consequently their return type is `void`.

10 Statements

This section gives the translation of ISO Standard C++ statements to IPR nodes.

10.1 Compound statement

Named mappings are initialized with blocks in function definitions. An IPR block is a statement and consists of a sequence of statements and optional sequence of handlers.

Standard C++ defines a compound statement as any brace-enclosed sequence of statements

compound-statement:
  { *statement-seq*$_{opt}$ }

statement-seq:

statement
statement-seq statement

The corresponding concrete IPR representation is `impl::Block`. Such a node is built with the member function

```
impl::Block* make_block(const ipr::Region&);
```

of the class `impl::Unit`. Suppose that we have to create nodes for the definition

```
int identity(int x) { return x; }
```

Then one would first create a block node for the body of the mapping associated with `identity`, and then fill in that block with sequence of statements as explained in sub-sections to follow.

```
impl::Block* body = unit.make_block(mapping->parameters, unit.Int());
mapping->body = body;
// fill in the body with add_stmt() as shown below
```

10.2 Expression statement

Most statements are actually expressions statements, which Standard C++ defines as

expression-statement:
expression$_{opt}$;

They are concretely represented with `impl::Expr_stmt`:

`unit.make_expr_stmt(`*expr*`)`

The case of *null statement*, *i.e.* an expression statement with missing expression, is handled by calling the (member) function `null_expr()` for the `Unit` object. An instance of null statement is the following fragment

```
while (*dst++ = *src++)
   ;
```

While statements are discussed in §10.4.1. Here, we just illustrate the representation an "empty" body:

```
Stmt* stmt = unit.make_expr_stmt(unit.null_expr());
```

10.3 Selection statement

A selection statement is any of the three kind of statements as defined by

selection-statement:
    `if (` *condition* `)` *statement*
    `if (` *condition* `)` *statement* `else` *statement*
    `switch (` *condition* `)` *statement*

They are concretely represented in IPR with `If_then`, `If_then_else` and `Switch` nodes, respectively.

Both `impl::If_then` and `impl::Switch` nodes are constructed in similar ways. They all require two arguments: the first being the condition and the second being the selected statement. Use `make_if_then()` to build a `impl::If_then` node, and `make_switch()` for a `impl::Switch` node. For instance, the fragment

```
if (lhs < rhs)
   return false;
```

may be translated as:

```
Expr* return_value = unit.make_literal(unit.Bool(), "bool");
unit.make_if_then(*unit.make_less(lhs, rhs),
                  *unit.make_return(*return_value));
```

An `impl::If_then_else` node requires three arguments: the condition, the then-branch statement and the else-branch statement. It is constructed through the (member) function `make_if_then_else()` of class `impl::Unit`.

10.4 Iteration statement

Standard C++ defines an iteration statement to be

iteration-statement:
    `while (` *condition* `)` *statement*
    `do` *statement* `while (` *condition* `)`
    `for (` *for-init-statement*$_{opt}$ *condition*$_{opt}$ `;` *expression*$_{opt}$ `)` *statement*

10.4.1 While statement

Constructing a `impl::While` node requires the condition node and the iterated statement node. For example, the following fragment

```
while (n != 0)
   n = process_line(n);
```

may be constructed with

```
impl::Var* n = ...
impl::Fundecl* processline = ...
// ...
Expr* cond = unit.make_not_equal(n, *unit.make_literal(unit.Int(), "0"));
impl::Expr_list* args = unit.make_expr_list(); // hold the arg-list.
args->push_back(n);
Expr* call = unit.make_call(processline, *args);
Stmt* stmt = unit.make_expr_stmt(*uni.make_assign(n, call));
Stmt* while_stmt = unit.make_while(cond, stmt);
```

10.4.2 Do statement

A do statement is constructed similar to a while statement. The (member) function to call is `make_do()` with the iterated statement and the condition expression as arguments, in that order.

10.4.3 For statement

A for statement is a curious and interesting statement. All its components are optional. A missing part is equivalent to either a null expression or a null statement.

A `For` node is created through the (member) function `make_for()` which takes four arguments, one for each components.

Let's first look at

```
for (int i = 0; i < N; ++i)
   stmt
```

In this case, the *for-init-statement* is a declaration. Therefore, we create a sub-region (of the active region) that will contain the declaration and we use the scope of that sub-region as the first argument for `make_for()`.

```
// the IPR node representing the for statement
impl::For* for_stmt = unit.make_for();

// Build the declaration for "i".
impl::Region* init_region = active_region->make_subregion();
impl::Var* i = init_region->declare_var(unit.make_identifier("i"),
                                        unit.Int());
i->init = unit.make_literal(unit.Int(), "0");

// set the for-init
for_stmt->init = &init_region.scope;

// Build the condition.
```

```
  for_stmt->cond = unit.make_less(*i, N);

  // the incrementation
  for_stmt->inc = unit.make_pre_increment(*i);

  // the body of the for-statement
  for_stmt->stmt = stmt;
```

If the declaration for the variable `i` was not limited to the for statement, *i.e.* if we had

```
  int i;
  for (i = 0; i < N; ++i)
    stmt
```

then we would not need to build a sub-scope for that variable. Rather, we would just build the declaration in the current scope:

```
// Build a declaration for "i",
Var* i = active_region->declare_var(*unit.make_identifier("i"), unit.Int());
impl::For* for_stmt = unit.make_for();
for_stmt->init = unit.make_assign(*i, *unit.make_literal(unit.Int(), "0"));
// the condition,
for_stmt->cond = unit.make_less(*i, N);
// the incrementation,
for_stmt->inc = unit.make_pre_increment(*i);
```

Another interesting case is when the *condition* in the for statement is actually a declaration. In that case, we build a sub-region (of the active region) and use it as the second argument to `make_for()`. Therefore the following fragment

```
  for (int i = 0; int j = N - i; ++i)
     stmt
```

may be translated by

```
  impl::For* for_stmt = unit.make_for();

  // Build the for-initialization part
  impl::Region* init_region = active_region->make_subregion();
  for_stmt->init = &init_region->scope;
  impl::Var* i = init_region->declare_var(*unit.make_identifier("i"),
                                          unit.Int());
  i->init = unit.make_literal(unit.Int(), "0");

  // The for-condition part
  impl::Region* cond_region = init_region->make_subregion();
  for_stmt->cond = &cond_region->scope;
```

```
impl::Var* j = cond_region->declare_var(*unit.make_identifier("j"),
                                        unit.Int());
j->init = unit.make_sub(N, *i);

// the incrementation part,
for_stmt->inc = unit.make_pre_increment(*i);

// and the body of the for-statement.
for_stmt->stmt = stmt;
```

Notice that the region containing `j` is a sub-region of the scope containing `i` and is the active scope till *stmt*.

10.5 Labeled statement

Standard C++ defines a labeled statement according to the grammar:

labeled-statement:
  identifier : *statement*
  `case` *constant-expression* : *statement*
  `default` : *statement*

All these three variants of labeled statements are uniformly represented in IPR through the node class `Labeled_stmt`. The label can be any IPR expression. Since a name is a expression a statement like

```
id:
   token = cursor - 1;
   // ...
```

may be represented in IPR as:

```
impl::Var cursor = ...;
// ...
impl::Minus* rhs = unit.make_minus(cursor,
                                  *unit.make_literal(unit.Int(), "1")));
impl::Assign* expr = unit.make_assign(*cursor, *rhs);
impl::Expr_stmt* expr_stmt = unit.make_expr_stmt(*expr);
impl::Idenifier* lbl = unit.make_identifier("id");
impl::Labeled_stmt* labeled_stmt = unit.make_labeled_stmt(*lbl, *expr_stmt);
```

Here a node created for the name `id` is used as the expression that labels the whole statement.

For a `case` label, the associated constant expression is used as the labeling expression. For example, for the program fragment

```
int line_count = 0;
// ...
switch (*cur) {
  case '\n':
     ++line_count;
     // ...
}
```

one might construct

```
impl::Var* linecount = ...
// ...
// literal used to label the case-statement
impl::Literal* nl = unit.make_literal(unit.Char(), "\\n");
impl::Labeled_stmt* stmt = unit.make_labeled_stmt
   (*nl, *unit.make_expr_stmt(*unit.make_pre_increment(*linecount)));
```

The `default` label is represented no different from ordinary labels. That is, one uses `unit.make_identifier("default")` as the labeling expression.

10.6 Jump statement

A jump statement is any of

jump-statement:
    `break ;`
    `continue ;`
    `return` *expression*$_{opt}$ `;`
    `goto` *identifier* `;`

Return-statements are built with the member function `make_return()` of `imp::Unit`. So, continuing with the `identity` function

```
body->add_stmt(unit.make_return(*x));
```

A break-statement is built with `make_break()`, a continue-statement is built with `make_continue()`, and a goto-statement is built with `make_goto()` taking the destination as argument. At the exception of the return statement, IPR nodes for jump statements have room to record the statements primarily affected by the control transfer. Consider the program fragment

```
char c;
int line_count = 0;
// ...
switch (c) {
  case '\n':
    ++line_count;
    break;
  // ...
}
```

Here is a corresponding IPR nodes construction:

```
// Build declaration for "c",
impl::Var* c = active_region->declare_var(*unit.make_identifier("c"),
                                           unit.Char());

// do the same for "line_count",
impl::Var* line_count = active_region->declare_var
   (*unit.make_identifier("line_count"), unit.Int());
line_count->init = unit.make_literal(unit.Int(), "0");

// ...
// Build the Block for the switch statement.
impl::Block* block = unit.make_block(active_region);
// Build the switch-statement node.
Switch* switch_stmt = unit.make_switch(*c, *block);

// Fill in the switch body,
Stmt* stmt = unit.make_expr_stmt(*unit.make_pre_increment(line_count));
Expr* lbl = unit.make_literal(unit.Char(), "\n");
block->add_stmt(unit.make_labeled_stmt(lbl, stmt));

  // Build the break statement
impl::Break* break_stmt = unit.make_break();
  // record the statement we're breaking from
break_stmt->stmt = switch_stmt;

  // put it in the body.
block->add_stmt(break_stmt);
// ...
```

10.7 Declaration statement

A declaration is a statement. As such, a declaration that appears at block scope shall be added to the sequence of statements that constitute the body of that block.

10.8 Try Block

Try blocks in Standard C++ come into various syntactic flavors.

try-block:
    `try` *compound-statement handler-seq*

function-try-block:
    `try` *ctor-initializer*$_{opt}$ *function-body* *handler-seq*

handler-seq:
    handler *handler-seq*$_{opt}$

handler:
    `catch ( ` *exception-declaration* ` )` *compound-statement*

In IPR, we do not have a separate node for try-block statement. Rather, we take the general approach that any block can potentially throw an exception; therefore any `Block` has an associated sequence of handlers. If that sequence is empty then it does not come from a try-block.

11 Parameterized declarations

In IPR, any expression can be parameterized. Parameterized expressions, and are uniformly represented with `impl::Mapping` nodes (see discussion in §9.1). Parameterized declarations, or template declarations in Standard C++ terminology, are declaration generators. For instance, consider the following generalization of the function `identity` from previous section:

```
template<class T>
  T identity(T x)
   return x;
```

equivalently written in XPR as

```
identity: <T: class> (x: T) T throws(...) = {
   return x;
}
```

It is clear that it is a named mapping which, when given a type argument T, produces a function declaration — named `identity<T>` – taking a `T` and returning a value of the same type. In a sense, it is a mapping of a mapping: the result of the first mapping is compilte-time, whereas the second is runtime; however the abstract representations are similar.

A named mapping is a declaration (`Named_map`). It has type represented by a `Template` node. Its initializer is a mapping of type `Template`.

11.1 Primary declaration generators

C++ template declarations can be divided into two categories: (a) primary templates; and (b) secondary templates.

A primary template is the most general form of a declaration generator. It indicates the type of the declaration it generates, and the number and sorts of arguments it accepts. A primary declaration generator participates in overload resolution (secondary declaration generators don't).

The notion of primary template declaration should not be confused with that of master declaration. A master declaration is the first declaration of an entity in a given scope. Primary declarations, on the other hand, may be repeated where permitted (for instance, at namespace scope).

```
  template<class T>
     struct Array {                          // #1
        // ...
     };

  template<class T>
     T sum(const Array<T>&);                 // #2

  template<class T>
     struct Array;                           // #3
```

`#1` is the first declaration of the template `Array<>`; so it is a master declaration. It is also the most general form of `Array<>` instance declaration generator; therefore it is also a primary template. In summary, `#1` is a master primary template declaration. On the other hand, `#3` is a redeclaration, therefore it is a non-master primary template declaration.

A node for a primary template declaration is built with the member function

```
  impl::Named_map* declare_primary_map(const ipr::Name&, const ipr::Template&);
```

of the enclosing user-defined type. A primary map does bookkeeping for various "administrative" information. The program fragment below builds the nodes for the representation of `#1`

```
#include <ipr/impl.H>
#include <ipr/io.H>
#include <iostream>

int main()
{
   using namespace ipr;
   impl::Unit unit;
   Printer pp(std::cout);

   // make the node that contains the body of the Array template
   impl::Mapping* mapping = unit.make_mapping(*unit.global_region());

   // declare the template type-parameter "T".
   impl::Parameter* T = mapping->param(*unit.make_identifier("T"),
                                       unit.Class());

   // set the type of the mapping.
   const ipr::Template* tt = unit.make_template(mapping->parameters.type(),
                                                unit.Class());
   mapping->constraint = tt;

   // build the decl for "Array", with same type.
   impl::Named_map* array = unit.global_ns->declare_primary_map
      (*unit.make_identifier("Array"), *tt);
   array->init = mapping;

   // "Array" uses the argument list "<T>".
   array->args.push_back(T);

   // and the body of the mapping is a class-expression
   impl::Class* body = unit.make_class(mapping->parameters);
   mapping->body = body;
   // set its name.
   body->id = unit.make_template_id(*array, array->args);

   pp << unit;
   std::cout << std::endl;
}
```

Every instance of `Named_map` keeps track of the (template) argument list it uses. That argument list is needed for the purpose of determining the most specialized version of a named mapping use.

11.2 Secondary named mappings

A secondary template declaration provides a specialized implementation for given sub-family of the primary template. Therefore, it may introduce more or fewer parameters than the primary template does. However, it must supply an argument list that meets the requirements (in number and kinds) stated by the primary template declaration. Secondary templates do not participate in overload resolution. Consider for example th previous `Array<>` declaration, continued as follows

```
  template<>
  struct Array<void*> {                    // #4
     // ...
  };

template<class T>
  struct Array<T*> : Array<void*> {        // #5
     // ...
  };
```

The declaration `#4` is an explicit specialization of `Array<>`, but it is *not* a template. Therefore, it must be represented as an ordinary class. On the other hand, `#5` is a secondary template declaration. It specializes `#1`, and uses the argument list `<T*>`.

Secondary named mappings are manifactured with the member function

```
impl::Named_map* declare_secondar_map(const ipr::Name&, const ipr::Template&);
```

of the enclosing user-defined type.

12 External representation

The persistent form of IPR nodes is expressed through the XPR syntax — for *eXternal Program Representation*. The XPR syntax is defined in the Reference Manual.

The header file `<ipr/io.H>` provides access to XPR output facilities. That header file defines:

- a basic XPR printer `ipr::Printer;`
- facilities to print *expressions*, *types*, *statements* and *declarations*.

The basic XPR printer takes a reference to an *std::ostream* — which is the stream into which it will insert the XPR syntax of whatever nodes it is asked to print. A typical construction is

```
#include <ipr/impl.H>
#include <ipr/io.H>
#include <iostream>
int main()
{
    ipr::impl::Unit unit;
    // construct IPR nodes ...
    // Print them out, starting from the global scope
    ipr::Printer printer(std::cout);
    printer << unit;
    std::cout << std::endl;
    // ...
}
```

References

[DRS04] G. Dos Reis and B. Stroustrup, *Internal Program Representation for The Pivot*, 2004.

[ISO98] ISO, *International Standard ISO/IEC 14882. Programming Languages — C++*, 1998.

[ISO03] ISO, *International Standard ISO/IEC 14882. Programming Languages — C++*, 2nd ed., 2003.