Extended abstract

The Pivot: A brief overview

Bjarne Stroustrup and Gabriel Dos Reis

bs@cs.tamu.edu, gdr@cs.tamu.edu

Abstract

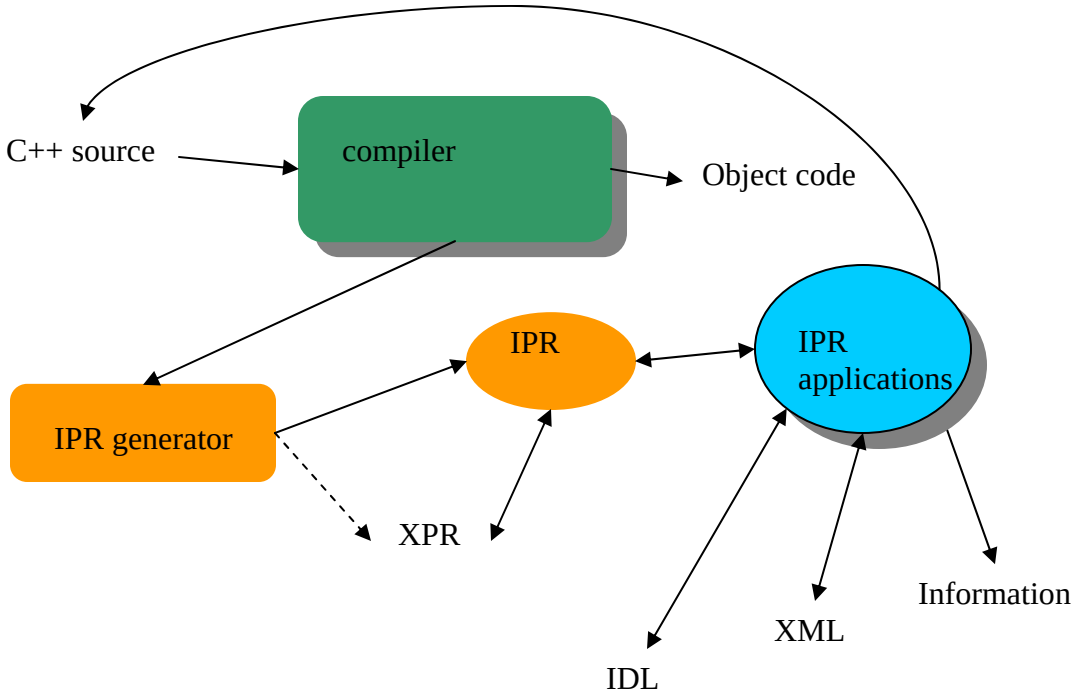
This paper introduces the Pivot, a general framework for the analysis and transformation of C++ programs. The Pivot is designed to handle the complete ISO C++, especially more advanced uses of templates and including some proposed C++0x features. It is compiler independent. The central part of the Pivot is a fully typed abstract syntax tree called IPR ("Internal Program Representation").

Introduction

There are lots of (more than 20) tools for static analysis and transformation of C++ programs, e.g. [Schordan, 2002] [Bagge, 2004] [Schupp, 2002] [Necula, 2004]. However, few – if any – handle all of ISO Standard C++ [ISO, 2003] [Stroustrup, 2000], most are specialized to particular forms of analysis or transformation, and few will work well in combination with other tools. We are particularly interested in advanced uses of templates as used in generic, programming, template meta-programming, and experimental uses of libraries as the basis of language extension. For that, we need a representation that deals with types as first-class citizens and allows analysis and transformation based on their properties. In the C++ community, this is discussed under the heading of "concepts" (see companion paper [Dos Reis, 2005]) and is likely to receive some language support in the next ISO C++ standard (C++0x) [Stroustrup, 2003a] [Stroustrup, 2003b]. This paper is an overview and will not go into details of the Pivot, the IPR, or our initial uses.

System organization

To get IPR from a program we need a compiler – only a compiler "knows" enough about a C++ program to represent it completely with syntactic and type information in a useful form. In particular, as simple parser doesn't understand types well enough to do a credible general job. We interface to a compiler in some appropriate (to a specific compiler) and minimally invasive fashion. A compiler-specific IPR generator produces IPR on a per-translation-unit basis. Applications interface to "code" through the IPR interface. So as not to run the compiler all the time and to be able to store and merge translation units without compiler intervention, we can produce a persistent form of IPR called XPR ("eXternal Program Representation")



Only a complete and correct C++ compiler can collect sufficient information for type-driven or concept-driven applications. From a compiler, we generate IPR containing fully typed abstract syntax trees. In particular, every use of a function name and operator is resolved to its proper declaration, all scope resolution is done, and all implicit calls of constructors and destructors are known.

We have IPR generators from GCC and EDG, so that the Pivot is not compiler specific. Early versions of this system (and its precursors) have been used to write pretty printers, generate XML for C++ source, CORBA IDL for C++ classes, and distributed programs using C++ source augmented with a library defining modularity. We do not assume the ability to produce information suitable for feeding IPR directly back into the internal formats of a compiler. Such a facility may exist for a give compiler, but to preserve the Pivot's independence of individual compilers such interfaces are not a priority.

IPR principles

The IPR ("Internal Program Representation") is a typed abstract syntax tree representation that retains all information from the C++ source (with exception of macro definitions). The IPR is compact, completely typed (every entity has a type, even types), representation with an interface consisting of abstract classes. The IPR has a unified representation so that its memory consumption is minimal. For example there will be only on node representing the type **int** and only one node representing the value **42** in a program that uses those two entities.

The IPR does its own memory management so users do not have to keep track of created objects. The IPR is arguably optimal in the number of indirections needed to access a given piece of information. The IPR is minimal in that it holds only information directly present in the C++ source. IPR can be annotated by the user and flow graphs can be generated. However, that's considered jobs for IPR applications rather than something belonging to the general framework itself. In particular, traversal of C++ code represented as IPR can be done in several ways, including "ordinary graph traversal code", visitors, iterators, or tools such as Rose [Schordan, 2003]. The needs of the application – rather than the IPR – determines what traversal method is most suitable.

IPR has two parts an immutable interface suitable and simplified for the large number of application that do not need to modify input source and a slightly more complicated interface for users that need to mutate the representation of an IPR module.

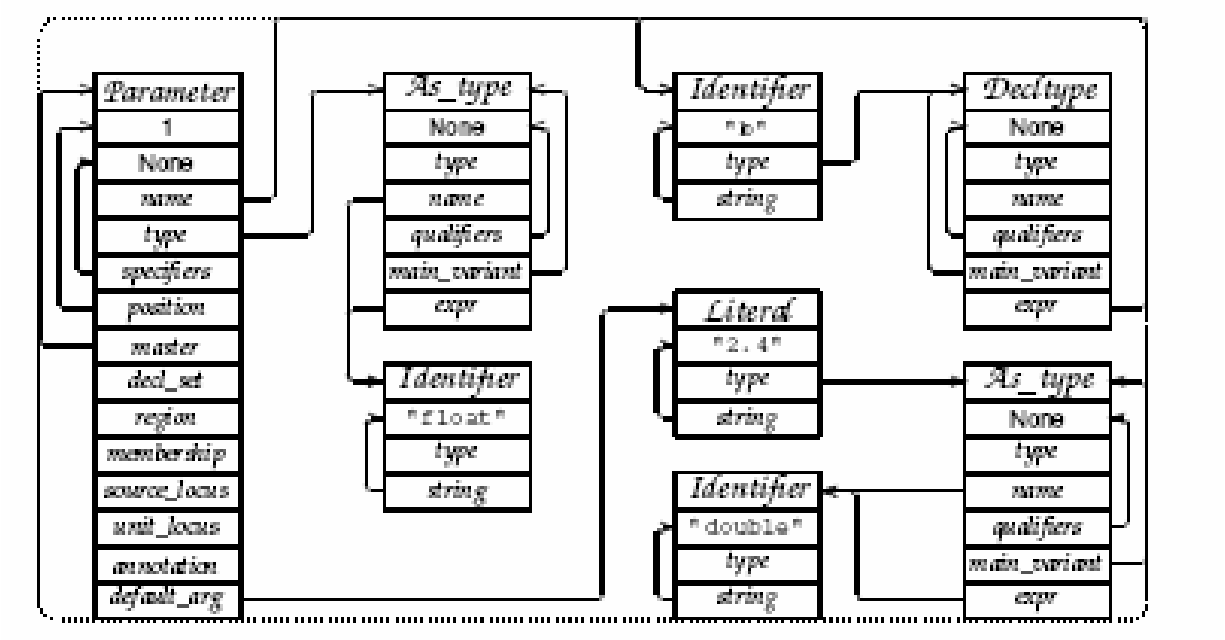
The IPR can represent both correct and incorrect (incomplete) C++ code and both individual translation units and merged units (such as a complete program). It is therefore suitable for both analysis of individual separately-compiled units and whole program analysis.

The IPR represents ISO C++ code. That implies that it can trivially be extended to represent C89 code and easily extended to common C++ dialects. However, since the initial aim of the Pivot is to look into high-level type-based and concept-based transformation, there is no immediate desire to extend it to cope with other languages with significantly different semantics, such as Fortran or Java.

Consider the declaration:

int foo(int a = 5, float b = 2.4)

The parameter **b** has type **float** and default parameter **2.4** (a floating-point literal of type **double**). Its IPR representation looks like:



Here, the named "fields" of the representation indicate member functions that supply the information; however stored. The arrows represent the values returned by such function for this example.

In IPR, a declaration is represented by three components:

(a) a name,
(b) a type, and
(c) an optional initializer.

The initializer for a parameter is its default argument value. The name of the declaration, represented as an **Identifier**, is an expression. That type is represented as **decltype(b)**, meaning "the declared type of b." That instance of **Decltype** in turn has a type (the "concept" of that type), not shown on the picture. The type (**float**) of the parameter declaration is a built-in type. Its representation is the identifier **float** interpreted as a type; basically **As_type("float")**. The default argument value, **2.4**, is a literal of type **double**. To maintain independence of any particular implementation, a literal is represented as a (string,type) pair. A declaration knows its enclosing declarative region, and the owner of that region (membership).

User programs can annotate IPR nodes. The IPR "remembers" the C++ source locations of its nodes.

XPR principles

The XPR ("eXternal Program Representation") is a compact (about the size of the original C++ source), human readable (details are "C++ like"), ASCII representation of IPR. It can be thought of as a persistent form of IPR or even as an object database for

IPR. XPR can be used as a transfer format between two different runs of the Pivot (saving us the bother of restarting a compiler) or two different implementations of the IPR. The XPR is designed to be parsed without a symbol table and with only a single token look ahead (in other words: fast). It is strictly prefix notation where types are concerned. Expressions and statements are rather C++ like.

An XPR example:

```
D : <| T : typename |>
        class : (B1 public, B2 public) {
                int16 : private typedef short;
                count : private int16;
                size : private int;
                element : T;
                #ctor : public (i1 : int16, i2 : int) count(0), size(1) { };
                foo : public (a : int <-{5}, b : float <-{2.4}) int
                {
                        a = b * a + element;
                        return a;
                };
        };
```

The corresponding C++ is:

```
template<class T>
class D : public B1, public B2 {
        private:
                typedef short int16;
                int16 count; int size;
                T element;
        public:
                D(int16 c, int s) : count(c), size(s) { }
                int foo(int a = 5, float b = 2.4)
                {
                        a = b * a + element;
                        return a;
                }
        };
```

Bindings to declarations and types are represented as annotations and kept near to, but separate from the human readable part.

Summary

The Pivot is a framework for analysis and transformation of ISO C++ programs with an emphasis on the higher levels of the C++ type system. It is very general and can be used

to interface arbitrary C++ analysis and transformation tools to a compiler (as long as the tool can manage without macro definitions).

References

- [Bagge, 2004] Otto Skrove Bagge, Karl Trygve Kalleberg, Magne Haveraaen, Eelco Visser: "Design of the CodeBoost Transformation System for Domain-Specific Optimisation of C++." Programs. http://www.codeboost.org/
- [Dos Reis, 2005] G. Dos Reis and B. Stroustrup: A high-level program representation for HPC. Companion paper.
- [Dos Reis,2005b] G. Dos Reis and B. Stroustrup: The design of the Pivot IPR, TAMU Technical Report, 2005.
- [ISO, 2003b] "The C++ Standard" (ISO/IEC 14882:2002). Wiley 2003. ISBN 0-470-84674-7.
- [Necula, 2004] George C. Necula, Scott McPeak, Shree Prakash Rahul, Westley Weimer: "CIL: Intermediate Language and Tools for Analysis and Transformation." http://manju.cs.berkeley.edu/cil/
- [Schordan, 2003] Markus Schordan and and Daniel Quinlan. A Source-to-Source Architecture for User-Defined Optimizations. In Proc. of the Joint Modular Languages Conference (JMLC'03), Volume 2789 of Lecture Notes in Computer Science, pp. 214-223, Springer Verlag, June 2003. (Rose).
- [Schupp, 2002] S. Schupp, D. P. Gregor, D. R. Musser, and S.-M. Liu. Semantic and behavioral library transformations. Information and Software Technology, 44(13):797 810, October 2002. (Simplicissimus).
- [Stroustrup, 2000] B. Stroustrup: "The C++ Programming Language (Special Edition)". Addison Wesley. Reading Mass. USA. February 2000. ISBN 0-201-70073-5.
- [Stroustrup,2003a] B. Stroustrup: "Concept checking - A more abstract complement to type checking". C++ standard committee. Paper N1510.
- [Stroustrup,2003b] B. Stroustrup, G. Dos Reis: "Concepts - Design choices for template argument checking" C++ standard committee. Paper N1522.
- [Stroustrup ,2005] B. Stroustrup: The Pivot XPR. TAMU Technical Report. 2005.
- [Schordan, 2003] Markus Schordan and and Daniel Quinlan. A Source-to-Source Architecture for User-Defined Optimizations. In Proc. of the Joint Modular Languages Conference (JMLC'03), Volume 2789 of Lecture Notes in Computer Science, pp. 214-223, Springer Verlag, June 2003. (Rose).