Supporting SELL for High-Performance Computing

Bjarne Stroustrup and Gabriel Dos Reis

Department of Computer Science
Texas A&M University
College Station, TX 77843-3112

Abstract. We briefly introduce the notion of *Semantically Enhanced Library Languages*, SELL, as a practical and economical alternative to special-purpose programming languages for high-performance computing. Then we describe the *Pivot* infrastructure for program analysis and transformation that is our main tool for supporting SELL. Finally, we outline how the IPR (The Pivot's *Internal Program Representation*) can be used to represent central notions of high-performance computing, such as parallelizable array operations. Our focus is on a broad exposition of ideas, rather than technical details[1].

1 Languages and libraries

For ease of programming, portability, and acceptable performance, we design and implement special-purpose programming languages for high-performance computing [15]. Alternatively, we can use a *Semantically Enhanced Library Language*. A SELL is a language created by extending a programming language (usually a popular general-purpose programming language) with a library providing the desired added functionality and then using a tool to provide the desired semantic guarantees needed to reach a goal (often a higher level semantics, absence of certain kinds of errors, or library-specific optimizations) [12]. This paper focuses on a tool, *The Pivot*, being developed to support SELLs in ISO C++ [11, 5] and its application to High-Performance Computing.

2 A brief overview of the Pivot

The Pivot is a general framework for the analysis and transformation of C++ programs. It 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*).

There are lots of (more than 20) tools for static analysis and transformation of C++ programs, e.g. [7, 2, 8, 6]. However, few — if any — handle all of ISO

[1] This is the "cut" or "abbreviated" version of this paper. For a full version, see http://www.research.att.com/bs/papers.html

Standard C++, 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* and is likely to receive significant language support in the next ISO C++ standard (C++0x) [13, 9, 14, 3]. We use the word concept to a designate collection of properties that describes usage of values and types. From the point of view of support for HPC — and for the provision of special-purpose facilities in general — a concept can be seen as a way of specifying new types with associated semantics without the modification of compilers or new syntax. That done, the SELL approach then uses the concepts as a hook for semantic properties beyond what C++ offers.

2.1 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, a simple parser doesn't understand types well enough to do a credible general job. We interface to a compiler in some appropriate 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*).

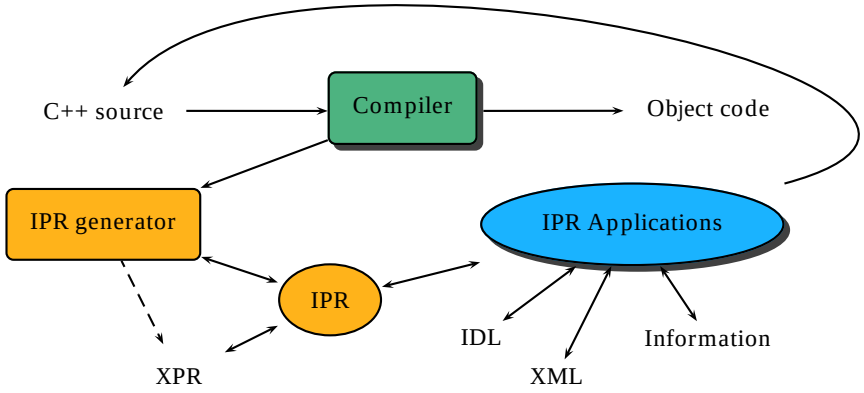

Fig. 1. An overview of *The Pivot* infrastructure

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 base our work on compilers, rather than on a simple parser because only a complete and correct compiler can collect sufficient information for type-driven and concept-driven applications. We have IPR generators from GCC and EDG, so that the Pivot is not compiler specific. The reason for preserving compiler independence is to maximize the portability of IPR-based tools. A tool that is built directly on a compiler's internal interface cannot easily be ported to another compiler. In fact, the interfaces to current C++ compilers' data structures for syntax and type information differ dramatically and most are de facto inaccessible for technical, commercial, or political reasons.

Early versions of this system (and its precursors) have been used to write pretty printers, generate XML for C++ source, CORBA IDL from C++ classes, and distributed programs using C++ source augmented with a library defining modularity.

The XPR is a compact and human readable ASCII representation of IPR. XPR can be used as a transfer format between two different runs of the Pivot or two different implementations of the IPR. The library implementing the IPR is elegant, compact, and efficient. It is just 2,500 lines of C++ to cope with all of C++, unify types (and literals and anything else we might want to unify), and manage memory.

2.2 IPR principles

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 one node representing the type `int` and only one node representing the integer value `42` in a program that uses those two entities. This minimalism (in time and space) is key to its use for large systems — million line programs are no longer rare.

The IPR does its own memory management so users do not have to keep track of created objects. It 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 core framework itself. In particular, traversal of C++ code represented as IPR can be done in several ways, including "ordinary graph traversal code", visitors [4], iterators [10, 11], or tools such as Rose [7]. The needs of the application — rather than the IPR — determines what traversal method is most suitable.

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 C code and 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.

User programs can annotate IPR nodes. An annotation is a (name,value) pair optionally attached to an IPR node by a Pivot application for its own uses. An annotation does not affect the way the IPR functions. The IPR "remembers" the C++ source locations of its nodes, so that a tool can refer back to the original source code.

3 High-level program representation for HPC

Type systems have been introduced in programming primarily for correctness and efficiency. For example, if we know at translation time that an operation involving read and write accesses is alias free, we can exploit that for generating efficient code. Some programming languages, notably FORTRAN, are designed to allow the compiler to assume the absence of aliases. Other general-purpose programming languages, such as C or C++, allow only a restricted set of type-based aliasing. For example a pointer of type `void*` can be used to access any kind to data, but a pointer of type `int*` cannot be effectively used to access data of type `double`.

A typeful programming discipline can help make programs both correct and efficient. Abstract representation of programs naturally enables symbolic manipulation. Here, we present an approach to correctness and performance based on IPR. We will use the notion of *parallelizable vector* operation as a running example.

Why C++? For the SELL approach we need a widely-used general-purpose language for our "host language". For type transformation and high-level work, we need a language that provides a flexible type system that can be used in a type-safe manner. For high-performance computing, we need a language that can efficiently use hardware resources and is available on high-end computers. For wide use, we need a non-proprietary and platform-neutral language.

3.1 A notion of parallelizable

Consider the classic operation

```
z = a * x + y;
```

where `a` is scalar; `x`, `y` and `z` denotes vectors, and the operations `*` and `+` are component-wise. It can be parallelized if we know that the destination `z` does not overlap with the sources `x` and `y` in a way that displays non-trivial data dependencies. That happens, for example, if we know no vector element has its address taken. For exposition purpose, we will simplify the notion of Parallelizable to a collection of types whose objects support the operation `[]` (subscription) but not `&` (address-of) on its elements. Consider the generic function

```
template<Parallelizable T>
void f(const T& v)
{
   double a = v[2];    // #1: OK
   double* p = &v[2];  // #2: NOT OK.
}
```

Line #1 is valid but line #2 is an error because it uses a forbidden operation. We generalized the standard notation `template<typename T>` which reads "for all `T`", to `template<Parallelizable T>` meaning "for all `T` such that `T` is is `Parallelizable`".

Concepts will almost certainly be part of C++0x. However, using IPR we can handle concepts without waiting for the C++ standards committee to decide on the technical details, see Section 3.3.

A programmer might use `Parallelizable` to constrain the use of a vector:

```
vector<double> v(10000);
// ...
f(v);  // f will use v as an Parallelizable (only)
```

Here we now know that `f()` will not use `&` on `v` even though the standard library vector actually allows that operation. We can use `f()` with its no-alias guarantee for any type that supports subscripting. For example we might use a STAPL [1] `pvector`:

```
pvector<double> vd(100000);
// ...
f(vd);
```

The concept checking allows no assumptions about types uses beyond what the concept actually specifies (here, a `Paralleizable` provides `[]`). In particular, no hierarchical ordering or run-time mechanisms are required.

Note that when defined in this way, `Parallelizable` requires no modification to C++0x or to any compiler. Furthermore, the use of `Parallelizable` is most likely to be composable with other facilities introduced as concepts – even if the facilities were developed in isolation.

Below, we will briefly present a high-level representation of C++ programs that support concept-based analysis and transformations.

3.2 Concepts in the IPR

A translation unit is represented as a graph with a distinguished root for the sequence of top-level declarations. In IPR, every entity in a C++ program is viewed as an expression possessing some type. So, types have types, which are called concepts. This becomes more useful, and maybe clearer, for a type variable as we find them in template parameter lists.

In Fig. 2, we have drawn a view of the representation of the declaration `Parallelizable T`. The declaration of the template-parameter `T` has type

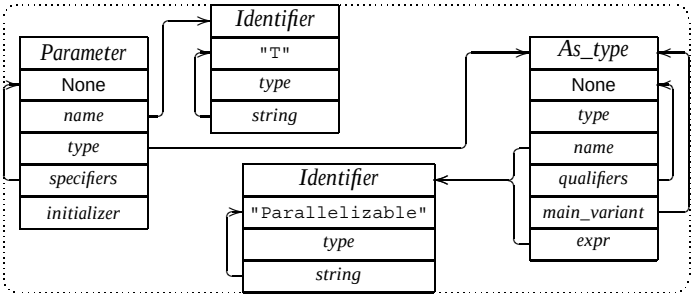

Fig. 2. IPR model `Parallelizable T`

`Parallelizable`. If we knew about the syntax and semantics of `Parallelizable`, that knowledge would be represented by a node referred to by the `type` field of the node with the identifier `"Parallelizable"`.

Note how `Parallelizable` fits into the IPR framework without modification or special rules. `Parallelizable` is simply a (deliberately trivial) example of what can be done with concepts in general.

Concepts are the basis for checking usage of types in templates, just like ordinary types serve to check uses of values in functions. Concept checking is done at two sites: (a) at template use site; and (b) at template definition site. If concept checking succeeds at both sites, then the template arguments are used (only) according to the semantics expressed in the concepts. In the particular case of `Parallelizable`, it means that no vector has its address taken, and consequently parallelization transformations can be safely applied.

3.3 Getting concepts into the IPR

How do we get concepts into our program? C++0x will most likely provide a way of specifying and checking concepts. That will provide a convenient handle for all concepts and for all SELL type-based analysis and transformation. For example:

```
concept Parallelizable<typename T> {
    // operations required by any Parallelizable type
    // only required operations will be accepted
    // for an object of a Parallelizable type
};
```

Once, the concept `Parallelizable` is part of the program, a Pivot application can operate based on its understanding of it. Note that this “understaning” can be extra-linguistic based on the tool builders knowledge of the semantics of the library of which `Parallelizable` is part.

However, what do we do if we don’t have a C++0x compiler that directly supports concepts? After all, C++0x won’t be fully specified for another couple of years. We could rely on annotations, pragmas, language extensions, etc., but

that has serious implications and costs. In particular, our programs would almost certainly not be composable with extensions defined and implemented by another group. The obvious alternative is to rely on convention: Traditionally, C++ programmers name template parameters to indicate their intended use. For example:

```
template<class Parallelizable>
void f(const Parallelizable& v)
{
    // operate on v according to Parallelizable rules
}
```

A Pivot application (tool) can easily recognize the type name Parallelizable and connect it to the definition of the concept Parallelizable as defined by the tool. From the point of view SELL and the Pivot, C++0x concepts is a significant convenience that provides a major advantage in notation and checking. However, it is only a (major) convenience because a Pivot-based tool can manipulate the IPR directly. For example, we could take code using the C++ standard library `accumulate`

```
template<class InputIterator, class T>
T accumulate(InputIterator first, InputIterator last,
             const T& init);
```

and transform every use into its equivalent parallel STAPL p-algorithm if (and only if) the STAPL requirements for its arguments are met. That is, the transformation takes place iff in addition to being an `InputIterator` the argument tyoe is a `BidirectionalIterator` or a `RandomAccessIterator`. This general approach to semantics-based transformation applies to all C++ standard algorithms described in terms of "abstract sequences".

The concept-based techniques rely critically on the use of templates, so that we can type template paraments with concepts to get a handle on their semantic properties. So, what do we do with code that doesn't use templates? Given an abstract syntax tree that represents a function declaration, we can transform it into a templated version and concept-check it. Consequently, we can check and transform a whole program as if it was fully templated.

4 Conclusion

The SELL, *Semantically Enhanced Library Language*, approach to supporting special-purpose languages can yield extension that are composable and portable. We presented our main tool for supporting the "semantic part" of that approach, *The Pivot*. The Pivot provides a general framework for analysis and transformation of C++ programs with an emphasis on high-level and type sensitive approaches. Our semantics-based analysis and transformation do not require modification to a host language and is minimally invasive to tool chains. It relies on a high-level program representation, the IPR, with emphasis on types

and concepts. Using the IPR we can perform analysis and transformation for high-performance computing (as well as other forms of computing) that traditionally required special-purpose languages or ownership of a specialized compiler and related tool chain.

References

1. Ping An, Alin Jula, Silvius Rus, Steven Saunders, Tim Smith, Gabriel Tanase, Nathan Thomas, Nancy Amato, and Lawrence Rauchwerger. STAPL: An Adaptive, Generic Parallel C++ Library. In *Proceeedings of the International Workshop on Languages and Compilers for Parallel Computation (LCPC)*, pages 193–208, Cumberland Falls, Kentucky, August 2001.
2. O. Bagge, K. Kalleberg, M. Haveraaen, and E. Visser. Design of the CodeBoost transformation system for domain-specific optimisation of C++ programs. In Dave Binkley and Paolo Tonella, editors, *Third International Workshop on Source Code Analysis and Manipulation (SCAM 2003)*, pages 65–75, Amsterdam, The Netherlands, September 2003. IEEE Computer Society Press.
3. Gabriel Dos Reis and Bjarne Stroustrup. Specifying C++ concepts. 2005. Accepted for publication at POPL06.
4. Erich Gamma, Richard Helm, Ralph Johson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1994.
5. International Organization for Standards. *International Standard ISO/IEC 14882. Programming Languages — C++*, 2nd edition, 2003.
6. Georges C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. CIL: Intermediate Language and Tools for Analysis and Tranformations of C Programs. In *Proceedings of the 11th International Conference on Compiler Construction*, volume 2304 of *Lecture Notes in Computer Science*, pages 219–228. Springer-Verlag, 2002. `http://manju.cs.berkeley.edu/cil/`.
7. M. Schordan and D. Quinlan. A Source-to-Source Architecture for User-Defined Optimizations. In *Proceeding of Joint Modular Languages Conference (JMLC'03)*, volume 2789 of *Lecture Notes in Computer Science*, pages 214–223. Springer-Verlag, 2003.
8. S. Schupp, D. Gregor, D. Musser, and S.-M. Liu. Semantic and behavioral library transformations. *Information and Software Technology*, 44(13):797–810, 2002.
9. Jeremy Siek, Douglas Gregor, Ronald Garcia, Jeremiah Willcock, Jaakko Järvi, and Andrew Lumsdaine. Concept for C++0x. Technical Report N1758=05-0018, ISO / IEC SC22 / JTC1 / WG21, January 2005.
10. Alexander Stepanov and Meng Lee. The Standard Template Library. Technical Report N0482=94-0095, ISO / IEC SC22 / JTC1 / WG21, May 1994.
11. Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, special edition, 2000.
12. Bjarne Stroustrup. A rationale for semantically enhanced library languages. In *Proceedings of LCSD'05*, October 2005.
13. Bjarne Stroustrup and Gabriel Dos Reis. Concepts — Design choices for template argument checking. Technical Report N1522, ISO / IEC SC22 / JTC1 / WG21, September 2003.
14. Bjarne Stroustrup and Gabriel Dos Reis. A Concept Design (rev.1). Technical Report N1782=05-0042, ISO / IEC SC22 / JTC1 / WG21, April 2005.
15. Gregory V. Wilson and Paul Lu, editors. *Parallel Programming using C++*. Scientific and Engineering Computation. MIT Press, 1996.