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. Please note that many of our considerations are practical in nature (relating to software engineering and economics) rather than fundamental from a scientific point of view. 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 [20]. These languages usually die young. Here, we briefly present a rationale for an alternative approach: *Semantically Enhanced Library Languages*, SELLs. 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). This paper focuses on a tool, *The Pivot*, being developed to support SELLs in ISO C++ [15, 7] and its application to High-Performance Computing.

1.1 Reasons for infant mortality

It is fun to design a new programming language. Doing the initial implementation and trying the new language with clever examples can be most exhilarating. However, bringing the implementation up to the level where users who care nothing about language design subtleties is plain hard work. Building supporting tools, such as debuggers and profilers, is hard work and not intellectually stimulating for most people who design programming languages.

[1] This is full version of a short paper presented to LCPC05

Real users also need basic numeric libraries, basic graphical facilities, libraries for interfacing with code written in other languages, "hand holding" tutorials, detailed manuals, etc. Doing each of those things once can be interesting and most educational, doing them all is tedious and often expensive. Porting the implementation, tool base, and key applications to new machines and compilers every half year or so is not only tedious, but also career death for many people. Basically, designing, implementing, maintaining and supporting a language is tremendously expensive. Only a large user community can shoulder the long-term parts of that.

The net effect is that on the order of 200 new languages are developed each year and that about 200 old languages become unsupported each year. "Language death" doesn't just happen to bad languages. For example, you can find a collection of 16 languages for high-performance computing in Wilson and Lu [20]: *Parallel Programming using C++*. Most have very appealing aspects, many are based on brilliant insights, all were supported by an enthusiastic research group, and all had years of stable funding. None are in major use today. None are supported by an organization outside the one that developed them. All but one are dead[2]. Interestingly, the live one (Charm++ [20]) is more of a library than a language.

In addition to the really ambitious language design projects, thousands of researchers work on research dialects and associated tools for their research. Such dialects are not built from scratch; instead, a compiler and key support tools are modified to serve the new dialect. Essentially all become unsupported upon graduation, funding expiration, tenure, promotion, transfer of maintenance responsibilities, change of fashion, change of any part of the tool chain, change of management, consolidation of IT operations, etc.

Some of these languages are designed for research only, but many are aimed at non-research use and most language designers harbor dreams of wide use for their languages. However, most of these new languages and dialects never see non-research use. The ones that do see non-research use are generally unloved by maintenance organizations. That is not just prejudice and unwillingness to learn or to change. There are perfectly good reasons for the lack of enthusiasm in maintenance organizations. For example, the supply of reasonably priced support personnel tends to be severely limited; good designers and good researchers (typically with PhDs) rarely want to become maintainers with an typical maintainer's salary, work conditions, and career prospects. Also, each new language and dialect has its own tool chain that needs to be kept current and in sync with other tools. The cost of doing so for a minor dialect is typically higher than for a major language — because the cost of the latter is amortized over millions of users. These reasons are often solid in economic and management terms, even though they can be heartbreaking for the proponents of a new language or dialect. For example the largest application using ML within AT&T was rewritten in a non-research language and so were the few uses of a very

[2] We'd love to be proven wrong on this; if you have a counter example, please tell us and we'll celebrate this exceptional success together.

interesting rule-based language R++ that can be seen as an early precursor of aspect-oriented programming [9].

1.2 Dialects

A popular “low-cost” approach to providing special-purpose facilities is to add pragmas, annotations, compiler options, “just a couple of extensions”, etc. to a popular language. This creates a dialect of a host language that can serve a specific user-group quite well. However, each such dialect needs its own variant of the compiler and similar tools and tend to be second priority (or worse) to the maintenance organization of the host language. Consequently, such a simple dialect is rarely supported for more than a single compiler or for a single platform. Rarely are such dialects as widely usable as the host language. This isolates the users of the dialect and limits the usefulness of their work.

Worse, two such dialects are rarely composable. A user must choose between “dialect A” and “dialect B”. For example, you might be able to get direct support for parallel programming or for data-base integration, but not both. The fundamental problem is that when two groups each create a dialect, they quickly end up on separate branches of a source code tree or in separate compilers.

Clearly, the ideal for support of special-purpose facilities includes composability with other special-purpose facilities and availability on every platform where the host language is used. It follows that support for special-purpose languages ideally fit within a framework that can accommodate a very general notion of "feature" rather than having the designer of each special-purpose feature provide separate facilities for accommodating syntactic and semantic extensions.

1.3 Language ideals

The prevalence of new languages and dialects in research together with their failure in production use lead to a most unfortunate chasm between researchers and developers. It is not uncommon that neither understand the other’s concerns, ideals, tools, and vocabulary. This can lead to misunderstanding, conflict, and mutual distrust. The difference in the languages used in research and in development is a significant barrier to technology transfer. Thus, language design can do serious harm — even when a design is technically sound and provides a vastly superior solution to a particular problem.

However, let’s be clear: The ideal solution to every major application problem is a special-purpose language designed specifically to meet the ideals of that application. For a given problem, we can always provide a superior solution in the form of a special-purpose language. We can tailor the semantics to the specific needs, we can adjust the syntax to be minimal and/or specialized for the culture of the programmers who will use it. From a narrow point of view, a good general purpose-language is at most second best. However, once we take tool costs, training costs, previous experience, cost of interacting with

other applications and groups, time from start of project to application delivery, etc. into account the balance dramatically shifts in the direction of the general-purpose language and well-known techniques.

So, a special-purpose language is the ideal for the individual application, but a general-purpose language has great advantages for maintenance, training, collaboration between diverse groups, tool chain availability, etc. What are the alternatives to a special-purpose language? Which tools and techniques can provide many of the advantages without suffering the problems? Many programmers and researchers still dream of a general-purpose language that is so expressive, elegant, and efficient that it can cover all application areas for all kinds of users without significant inconveniences. Some vigorously insists that their favorite language actually is such a language. However, no current language in major use fits this ambitious profile, and we don't see a research language that could possibly grow into that position in the short-to-medium term (e.g., within the next 15 years). Consequently, we suggest an alternative approach: semantically enhanced libraries embedded in a widely-used general-purpose language.

The rest of this paper describes the tools built to support this approach for C++ libraries.

2 A brief overview of the Pivot

The Pivot is 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*).

There are lots of (more than 20) tools for static analysis and transformation of C++ programs, e.g. [11, 3, 12, 10]. However, few — if any — handle all of ISO Standard C++ [7, 15], 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) [17, 18, 16, 13, 19]. You can think of a concept as the type of a type. 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.

This paper is an overview and will not go into details of concepts, the Pivot, the IPR, or our initial uses.

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 (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*).

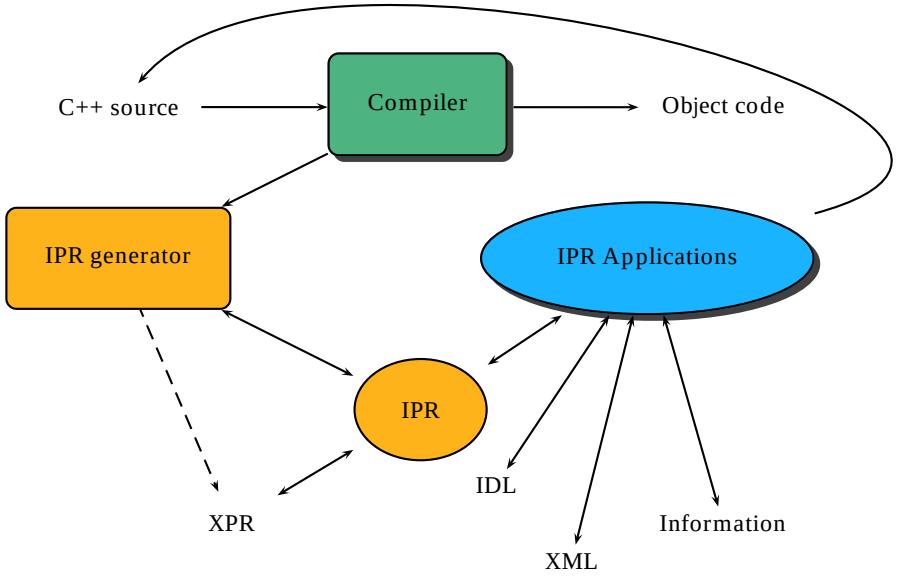

Fig. 1. An overview of *The Pivot* infrastructure

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 from 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. 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.

2.2 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 integer value `42` in a program that uses those two entities. This minimalism (in time and space) of the IPR 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. 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 core framework itself. In particular, traversal of C++ code represented as IPR can be done in several ways, including "ordinary graph traversal code", visitors [5], iterators [14, 2, 15], or tools such as Rose [11]. The needs of the application — rather than the IPR — determines what traversal method is most suitable.

The IPR library has two parts:

1. an immutable interface suitable and simplified for the large number of application that do not need to modify input source
2. 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 C [8, 6] 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 argument `2.4` (a floating-point literal of type `double`). Its IPR representation is displayed in Fig. 2.

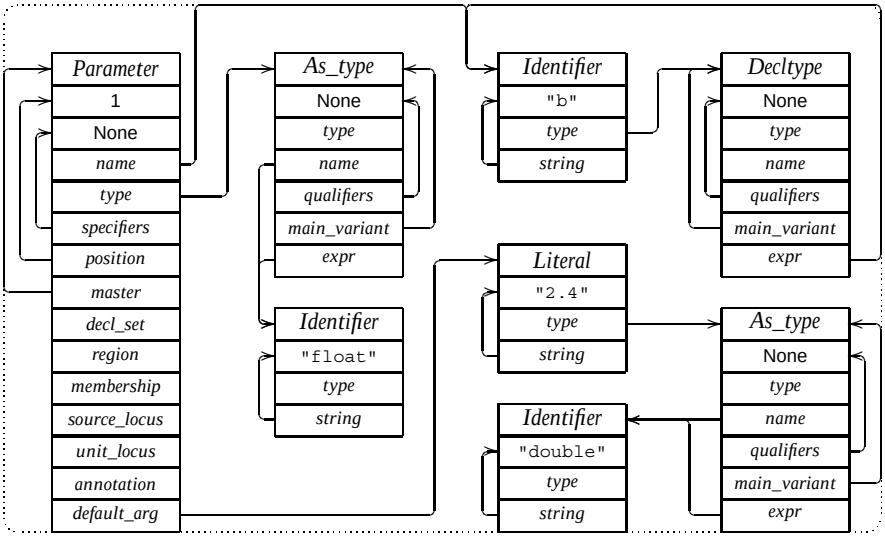

Fig. 2. IPR for the parameter declaration `float b = 2.4`

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 four essential and two optional components:

1. a name,
2. a scope (in which the name is declared)
3. a type, and
4. a set of specifiers.
5. an optional initializer
6. an optional set of annotations

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). 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.

User programs can annotate IPR nodes. The IPR "remembers" the C++ source locations of its nodes, so that a tool can refer back to the original source code.

2.3 XPR principles

The XPR (*eXternal Program Representation*) is a compact (about the size of the original C++ source) and human readable 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.

Here is an XPR example:

```
D : <T : typename>
   class : (public B1, public B2) {
      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;
      };
};
```

Obviously we use the *name : definition* notation. The details are "C++ like" while avoiding grammar ambiguities; for example, `< ... >` defines template parameters, `( ... )` function parameters, and `<-` means initialize. 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.

2.4 Pivot 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, such as templates, overloading, specialization, and concepts (proposed for C++0x). 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). The Pivot is compiler neutral and minimally invasive into a tool chain.

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.

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 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 programming discipline that effectively uses the type system 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 using a high-level, fully typed abstract representation of ISO C++ programs developed as part of *The Pivot* framework. 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 operating system 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. For instance, in 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".

We use the word *concept* to a designate collection of properties that describes usage of values and types. Basically, a concept is the type of a type. There are proposals to integrate concepts into the C++ programming languages [17, 18, 16, 13, 19]. However, using IPR we can handle concepts without waiting for the C++ standards committee to decide on the technical details 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.

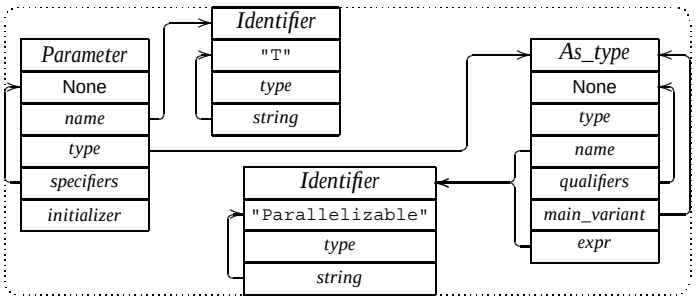

Fig. 3. IPR model `Parallelizable T`

In Fig. 3, we have drawn a view of the representation of the declaration `Parallelizable T`. The declaration of the template-parameter `T` has type `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 it is guaranteed that template arguments are used (only) according to the semantics expressed in the concepts [4]. 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
};

template<Parallelizable T>
void f(const T& v)
{
    // operate on v according to the rules
    // specified for Parallelizable
}

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

Once, the `Parallelizable` concept 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 as noted in 1.2, 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 the rules of Parallelizable
}
```

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& initalizer);
```

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

We presented a rationale for the SELL, *Semantically Enhanced Library Language*, approach to supporting special-purpose languages, arguing that this approach has practical and economic advantages compared to conventional approaches to providing special-purpose facilities. In particular, SELL 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. Matthew H. Austern. *Generic Programming and the STL*. Addison-Wesley, October 1998.
3. 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.
4. Gabriel Dos Reis and Bjarne Stroustrup. Specifying C++ concepts. 2005. Accepted for publication at POPL06.
5. Erich Gamma, Richard Helm, Ralph Johson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1994.
6. International Organization for Standards. *International Standard ISO/IEC 9899. Programming Languages — C*, 1999.
7. International Organization for Standards. *International Standard ISO/IEC 14882. Programming Languages — C++*, 2nd edition, 2003.
8. Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, 1988.
9. Diane Litman, Peter F. Patel-Schneider, and Anil Mishra. Modeling Dynamic Collections of Interdependent Objects Using Path-Based Rules. In *Proceedings of the 12th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA-97)*. ACM Press, October 1997. `http://www.research.att.com/sw/tools/r++/`.
10. 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/`.
11. 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.
12. S. Schupp, D. Gregor, D. Musser, and S.-M. Liu. Semantic and behavioral library transformations. *Information and Software Technology*, 44(13):797–810, 2002.
13. 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.
14. Alexander Stepanov and Meng Lee. The Standard Template Library. Technical Report N0482=94-0095, ISO / IEC SC22 / JTC1 / WG21, May 1994.
15. Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, special edition, 2000.
16. Bjarne Stroustrup. Concept checking — A more abstract complement to type checking. Technical Report N1510, ISO / IEC SC22 / JTC1 / WG21, September 2003.
17. Bjarne Stroustrup and Gabriel Dos Reis. Concepts — Design choices for template argument checking. Technical Report N1522, ISO / IEC SC22 / JTC1 / WG21, September 2003.
18. Bjarne Stroustrup and Gabriel Dos Reis. Concepts — syntax and composition. Technical Report N1536, ISO / IEC SC22 / JTC1 / WG21, September 2003.
19. Bjarne Stroustrup and Gabriel Dos Reis. A Concept Design (rev.1). Technical Report N1782=05-0042, ISO / IEC SC22 / JTC1 / WG21, April 2005.
20. Gregory V. Wilson and Paul Lu, editors. *Parallel Programming using C++*. Scientific and Engineering Computation. MIT Press, 1996.