A rationale for semantically enhanced library languages

Bjarne Stroustrup
Department of Computer Science
Texas A&M University
College station, TX-77843
and AT&T Labs – Research
bs@cs.tamu.edu

Abstract

This paper presents the rationale for a novel approach to providing expressive, teachable, maintainable, and cost-effective special-purpose languages: A *Semantically Enhanced Library Language* (a *SEL language* or a *SELL*) is a dialect created by supersetting a language using a library and then subsetting the result using a tool that "understands" the syntax and semantics of both the underlying language and the library. The resulting language can be about as expressive as a special-purpose language and provide as good semantic guarantees as a special-purpose language. However, a SELL can rely on the tool chain and user community of a major general-purpose programming language. The examples of SELLs presented here (*Safe C++*, *Parallel C++*, and *Real-time C++*) are based on C++ and the Pivot program analysis and transformation infrastructure. As part of the rationale, the paper discusses practical problems with various popular approaches to providing special-purpose features, such as compiler options and preprocessors.

1 Introduction

We often need specialized languages. Researchers need to experiment with new language features, such as concurrency features [24], facilities for integration with databases [5], and graphics [4] . Developers can sometimes gain a couple of orders of magnitude reductions in source code size with corresponding reductions in development time and defect rates, by using such special-purpose languages in their intended domains. Unfortunately, such special-purpose languages are typically hard to design, tedious to implement, expensive to maintain, and — despite their obvious utility — tend to die young.

Using a (special-purpose) library is an obvious alternative to a special-purpose language. However, a library cannot express or exploit semantic guarantees beyond what its host language provides. The basic idea of *Semantically Enhanced Library Languages* (SEL Languages or simply SELLs) is that when augmented by a library, a general-purpose language can be about as expressive as a special-purpose language and by subsetting that extended language, a tool can provide about as good semantic guarantees. Such guarantees can be used to provide better code, better representations, and more sophisticated transformations than would be possible for the full base language. For example, we can provide support for parallel operations on containers as a library. We can then analyze the program to ensure that no undesirable access to elements of those containers occurs — a task that could be simplified by enforcing a ban of languages features that happened to be undesirable in this context. Finally we can perform high-level transformations (such as parallelizing) by taking advantage of the known semantics of the libraries.

Like a library, a SELL can benefit from the extensive educational, tools, and library infrastructure of the base language. Therefore, the cost of designing, implementing, and using a SELL is minuscule compared with a special-purpose language with a small user base. Examples will be based on ISO standard C++ supported by the Pivot infrastructure for program analysis and transformation (§5.2). The focus will be on templates because they provide the key mechanism for statically type-safe expression of advanced ideas in C++.

What is called a "special-purpose language" here is often called a domain-specific language (e.g. [10]). Distinctions can be made between the two terms, but none that appear relevant to the discussion here, so please consider those two terms as equivalent in this context.

The organization of this paper is

1. Introduction
2. State some ideals for support of software development and maintenance.
3. Present some of the — usually fatal — problems that face new programming languages.
4. Discuss a few alternative approaches, such as dialects and macro languages.
5. Focus on the SELL approach and the way it can be supported in C++ using the Pivot.
6. Sketch the design of a few SELLs: type-safe C++, Parallel C++, and Real-time C++.
7. Conclusions

Submitted to LCSD'05

2 Ideals

For every specific problem area, we can design a special-purpose language that exactly matches the desired syntax and semantics of the domain and the desires of the programmers that will use that language. In an ideal world, no general- purpose language can match such a special-purpose language when applied in its specific problem area. When a special-purpose language has been done perfectly, there is a one-to-one correspondence between the fundamental concepts of the application domain and the language constructs. Given that, the language constructs can be minimal and directly reflect the terminology of the field as found in common use and major textbooks.

This is not a new ideal. Fortran did a good job at that task for arithmetic in the 1950s and COBOL successfully attacked the business processing needs of the time. Since then, thousands of languages have been designed for specific domains and almost as many have been designed to try to be able to effectively express that ideal for less specific domain. Lisp and Simula originate the two main approaches to more directly express application domain concepts directly in code: the functional and object-oriented approaches. In these languages, and in their numerous offspring, a set of concepts is represented as a library of related functions or classes. In such general-purpose and near-general-purpose languages the ideal of the perfect language for the task takes the form of libraries.

What do we expect from a well-designed special-purpose language? Concise notation is the beginning. Consider a simple, common, and useful example:

```
A = k*B + C
```

First note the algebraic notation using operators. Notation is important for concise expression of key ideas in a community. This particular notation is based on almost 400 years of history in the mathematics / scientific community.

Essentially all languages can handle `A=k*B+C` when the variables denote scalar values, such as integers and floating point numbers. For vectors and matrices, things get more difficult for a general-purpose language (that doesn't have built-in vector and matrix types) because people who write that kind of code expect performance that can be achieved only if we do not introduce temporary variables for `k*B` and `k*B+C`. We probably also need loop fusion (that is doing the element * , +, and = operations in a minimal number of loops). When the matrices and vectors are sparse or we want to take advantage of known properties of the vectors (e.g. `B` is upper-triangular), the library code needed to make such code work pushes modern general purpose language to their limit [17, 23] or beyond — most mainstream languages can't efficiently handle that last example. Move further and require the computation of `A=k*B+C` for large vectors and matrices to be computed in parallel on hundreds of processors. Now, even an advanced library requires the support of a non-trivial run-time support system [1]. We can go further still and take advantage of semantic properties of operations, such as "remembering" that C was the result of an operation that leaves all its elements identical. Then, we can use much simpler add operation that doesn't involve reading all the elements of `C`. For other examples, preceding the numerical calculation with a symbolic evaluation phase, say doing a symbolic differentiation, can lead to immense improvements in accuracy and performance. Here, we leave the domain where libraries have been considered useful. Reasoning like that and examples like that (and many more realistic ones) have led to the creation of a host of special-purpose languages for various forms of scientific calculation [24].

So, the ideal notation offered by a general purpose language is just the beginning. It can be the basis for comprehension, for fast compilation, for performance (exploiting type information and semantic properties), for reasoning about programs (by the implementation or associated tools), for programmer productivity, for making facilities accessible to professionals who need to program in their field of expertise, yet don't want to become professional programmers (e.g., physicists, engineers, animators, and graphical designers). Finally, the clarity of the code can greatly ease maintenance.

Note that the ideals and strengths of special-purpose and general-purpose languages can conflict. By definition, a general-purpose language aims at allowing the programmer to express just about anything. On the other hand, a special-purpose language gains much of its strength from allowing a programmer to express only what makes sense in its specific domain. When it comes to program analysis and optimization, this is a great strength of a special- purpose language. For example, if an optimizer tries to do a symbolic differentiation of a program in a language focused exclusively on scientific computation, it does not have to worry about a programmer trying to differentiate the draw function of a graphics system.

Convenient graphical interfaces are often associated with special-purpose languages. They can be used as an extreme example of direct representation of ideas or as a special- purpose language. However, such interfaces can be used to equal effect for code in a general-purpose language, so GUIs will not be examined further here.

3 Problems

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, it is plain hard work to bring the implementation up to the level needed for users who care nothing about language design subtleties. Building supporting tools, such as debuggers and profilers, is hard work and not intellectually stimulating for most people who design programming languages. 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 or repeatedly is tedious and often expensive. Porting the implementation, tool base, and key applications to new machines, platforms, and compilers repeatedly 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 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 *Parallel programming using C++* [24]. 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[1]. Interestingly, the one survivor (Charm++) is more of a library than a language.

In addition to the really ambitious language design projects, thousands of researchers work on 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 (or claim to be), but many are aimed at non-research use (or claim to be) 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, 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 a typical maintainer’s salary, work conditions, and career prospects.

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 was the few uses of a very interesting rule-based language R++ that can be seen as an early precursor of aspect-oriented programming [11].

Tool chain problems don’t just happen to “Mom&Pop languages”. I have seen major organizations abandon Ada for just this reason. Similarly, education can be a major problem. If a language isn’t taught in universities (or only in a few schools), good programmers become scarce and most organizations cannot afford to re-train new hires. Furthermore, new programmers are sometimes overly impressed by their favorite language and resists training. I have seen organizations abandon Fortran for that reason. The two effects are mutually reinforcing.

However, most special-purpose languages, proprietary dialects, etc. never get a large enough user base and tool set to worry about decline. Most minor and research languages simply never gain the tool support and availability on a wide range of platforms that users of mainstream languages take for granted. Unless a new language is really a minor dialect of an existing language, almost all of the design and implementation effort is recreating facilities — such as debuggers, profilers, database interfaces, and GUI interfaces — that tend to lie outside the main interest of the language designers. This repettetive reconstruction of “standard facilities” provided for other languages breeds lots of “good little ideas” as people add improvements. Unfortunately, such “little improvements” tend to further isolate users. Since “further isolates” can be read as “locks-in users” as well as “provides better support than the competition”, there is often little resistence to gratuitous replication and incompatibility. Compatibility is just hard work, and typically unrewarding.

How many users does it take to sustain an infrastructure? Of course, that depends on a lot of things, but generally it requires more people than work on a single application. In fact, it typically takes at least a small company. That is more — often significantly more — people than it took to create the initial design and implementation of a language. If — as is usual — these people have to be paid from the revenues from sales and teaching, a special-purpose language now comes under pressure to become more widely useful. That is, the special-purpose language starts to offer facilities for general computation, general data structures, access to “external systems”, database facilities, graphics facilities, etc. The result can be summarized as “Every special-purpose programming language wants to grow up and become a general-purpose programming language.” Typically, this is a precursor to “language death” (becase of instability, lack of design focus, and added cost) or to a retreat into a commercially viable niche that covers only a small part of the special-purpose language’s natural application domain. This withdrawl is often accompanied with a lot of commercial hype and a tendency to hide and obscure genuine technical information.

Many (probably most) special-purpose languages suffer from “edge effect” problems. The “edge effect” (also more evocatively known as the “falling off the cliff” effect) comes when a programmer needs to do something that isn’t supported by the special-purpose language. For example, a programmer using a language for specifying interactive graphics might want to say “when viewed from a sufficient distance, groups of objects may be considered one object”. The graphics system could have provided such a feature, but in this case it didn’t (and the difference in real-time response was about a factor of 100). What does the programmer do? By definition, every special-purpose language has such “edges”. For students and novices, the effect can be a nuisance; for professionals working on large projects (such as the airline control application from which this graphics example was chosen), the result can be the abandonment of the special-purpose language in favor of an alternative, such as a graphics library written in a general-purpose language. But what does a programmer do if changing tools isn’t an option? In a “pure” special-purpose language, a new primitive operation or object must be added. That’s not something every application programmer can do because it may effect the basic model of the special-purpose language. I have seen the time for adding a simple feature vary from one day (ask a local expert and wait for the overnight tool build) to half a year (wait for the

[1] I’d love to be proven wrong on this, so if you have a counter example, please tell me and we’ll celebrate this exceptional success together.

next release) or more. This kind of delay can kill a project, so it must be considered among the risks when choosing or designing tools. For a library — and for any tool that allows a programmer to add code written in a general-purpose language — the problem is minor.

The final nail in the coffin of many special-purpose languages is that once it is designed and in use, it is relatively easy to "emulate" its facilities in a general-purpose language. Often, the value of a special-purpose language is not really in the language implementation or its particular syntax (though programmers can be passionately devoted to a syntax). The value is in the design, the programming model, the techniques for use, and possibly some special algoritms or data structures sustaining applications. Typically, those special-purpose language "implementation details" can be separated from the language and used directly from a general-purpose language. This is all the easier because these key components are written in some general-purpose language. All that is needed for their direct use is a nice programming interface in that general-purpose language. The definition of "nice" will reflects the experience gained from the use of the special purpose language.

Please note that a language is rarely "killed" by any one of the problems mentioned. Typically, the language succumb to a combination of problems. Also, this list is not intended to be complete or necessary "fatal": some special-purpose languages do survive and some fail because of reasons not listed here. An exhaustive list of problems probably couldn't be compiled, and if it could it would be beyond the scope of this paper.

3.1 Case study: R++

A detailed study of a few hundred new languages to provide solid evidence for the observations made here would be useful. However, I doubt it would dampen the enthusiasm for designing new languages. Here, I'll just present one small example, and then proceed to an alternative approach to providing new facilities for programmers.

R++[11] is an unrecognized precursor to aspect-oriented programming. Basically, it is an extension of C++ in which you can define actions and triggers for actions. For example, a retirement policy can be associated with an `Employee` class like this:

```
rule Employee::retirement_policy {
     age>=65 && status!=retired
=>
     cout << name << " must retire...";
}
```

This is simple enough to be easy to teach. Furthermore, the implementation was a small enough increment on C++ that it was relatively easy to maintain. Since R++ is a superset of the general-purpose language C++ there are no edge effects. It was used in a reasonably large telecom operations system application. Tutorials, academic papers, manuals, experience reports, implementation, etc. were provided. You can find them on the web [11].

For all practical purposes, it died in 1996. The reasons were basically that the porting and training costs were too high compared to the benefits. What do I mean by dead? Completely unused? Not necessarily. Ever so often, I see a reference to R++ and I'd be surprised if there wasn't a project somewhere using it. Probably, there are also a couple of research groups trying it out. However, despite ideas that appear fundamentally sound, despite avoiding edge effects by being embedded in a general-purpose language, and despite having an implementation that did sustain a major application, R++ still suffered many of the various problems mentioned in this section and failed to gain major use outside its originating organization.

4 Alternatives

So, in most cases, designing a new language is not an economically viable solution to the problem of how to provide special-purpose facilities. A language often looks good for a few years but maintenance, porting, education, etc. is too expensive and the result is death or at best stagnation of the tool chain and the user community. As a technical / economical choice, designing a new language most often is a mistake. Most language design efforts soak up resources reinventing a few wheels and then die having provided a poor return on investment. The resources could have been better spent on improvements to an existing major language and its libraries and tools. Furthermore, most new languages divide a community by creating barriers to communication of new ideas and not infrequently by generating hype that trigger language wars and distrust of new ideas. Not all of the problems are the fault of the new language and new ideas must be explored and exploited. So, what else can we do to bring the ideal of direct expression of ideas in code into wider use?

So, let's assume that we are in one of the many situations where designing a new language is likely to be uneconomical and to have undesirable effects on the spread of ideas. What alternatives do we have when our task is to provide programmers with improved tools for expression ideas in code?

Here are some popular approaches:

1. Compiler options and pragmas
2. Libraries
3. Preprocessed languages
4. Dialects

Each can be an effective approach in some cases and each has been used in ways that have been deemed successful. Here, we must consider their fundamental and practical strengths and weaknesses.

These are not the only possible approaches. For example, one might consider:

1. Dynamically typed languages
2. A new, more general, general-purpose language

Dynamically typed languages are not considered here. The the main reason is an interest in compile-time guarantees. Basically, dynamically -typed languages constitutes a different world from the statically typed world that I focus on here. Dealing with that world is beyond the scope of this paper.

One might consider building a new general purpose language providing facilities that are so complete that every special-purpose language can be expressed directly through the mechanism of the general purpose language. That's one of the holy grails of general-purpose language design. In fact, over the last 30 years or so, there has been a stream of such langugages offering facilities for defining extended syntax (e.g. through embedded parsers) and associating semantics with the newly defined constructs. Such languages are also beyond the scope of this paper. Part of the reason is that providing such a language is beyond the means of most organizations needing a special-purpose language. Another problem is that (ironically) such languages themselves suffer from the problems of being special-purpose languages with small user communities and insufficient support. The success rate for general-purpose languages is even lower than the rate for special-purpose languages.

4.1 Compiler options and pragmas

People who add compiler options and/or pragmas rarely think that as language design. In particular, (in the C and C++ worlds) a `#pragma` can be ignored by a compiler. However, every new `#pragma` and compiler option introduces a new dialect. It is something to consider when building a system, when specifying a system configuration, when porting a system, when documenting a system, and when trying to understand application code. Assume for a moment that options and `#pragmas` are not used for back-door language extension. Then, they are simply insufficient for doing anything really interesting in the direction of better expression of ideas. Most special-purpose languages require additions. Also, they often require restriction of use of certain undesirable language features. That makes compiler options a too crude a mechanism. Options tend to apply indiscriminately; for example, we might want to eliminate the use of `goto`. However, the option will then eliminate all `gotos` — even the acceptable ones for breaking out of loops in a highly optimized matrix implementation and the essential ones in implementation of the state machines generated from a high-level modeling library/language. What is needed is to distinguish between uses of an undesirable language feature in user code and their use in the implementation of trusted components. Compiler options are best left for conventional uses, such as backwards compatibility switches; `#pragmas` are best avoided.

4.2 Libraries

Libraries can provide expressive power and notational convenience that approximate that of built-in language features. However, it is hard to ensure consistent use of a library (or a set of libraries). It is even harder to ensure consistent use of a subset of a library when — as is common — too much has been bundled into a single unit of distribution. Other language features can interfere with what a library attempts to achieve. The C++ standard library is a classical example. It provides well-behaved containers, but some programmers use arrays instead and thereby prevents any meaningful guarantees to be made for the program as a whole.

When ambitious in what they try to achieve in terms of generality or performance, libraries can become very elaborate and brittle. For example, some C++ template metaprogramming libraries aiming at very general support for high-performance numerical computation reach their goal at the cost of complete obscurity of implementation details that becomes visible to users during debugging. Often, a library breaks the zero-overhead principle in search for generality.

A library cannot, by itself, eliminate basic problems with host language semantics. For example, in C and C++, aliasing problems persists so that a library cannot provide guarantees needed for confidence, transformations, and optimizations. Often, a library is (at least partially) defined in terms of its implementation; it is not specified as an entity separate from its host language implementation. This is not a fundamental problem, but it is a common problem, and often a serious one in comparison to a special-purpose language.

4.3 Preprocessed languages

Generating code from a higher-level language into a lower-level one has been popular for decades. For example early C compilers generated assembly code; early C++ compilers generated C code; GUI builders, CAD systems, IDL processors, modeling languages, etc., generate code in languages such as C, C++, Java, C#. That is, the language source is preprocessed into a host language. The resulting languages and language processors are referred to by many names, such as preprocessors, macros, genrators, wizards, builders, and meta-languages. One way of distinguishing an implementation of a language implemented by such techniques from a facility defined by such translation techniques is whether you can ever get an error message from the target language compiler. If you can it's a preprocessor; if not it's a compiler. For example, by that criteria, the original C and C++ translators (into assembler and C) were compilers whereas Ratfor, C macros, and Microsoft "wizards" rely on preprocessors. C++ templates are "right on the edge" in that they receive some compiler support (and will recieve significantly more in the future: concepts [21, 20]). However, compiler error messages sometimes fail to refer to the original template source and often do so spectacularly badly. In consequence, some programmers consider templates "like macros" and avoid them; many more avoid uses they consider nontrivial. Here, we consider preprocessed languages, rather than abstraction facilities integrated within a language.

The language (generator, macro-language, modeling language, whatever) defined by a preprocessor becomes yet another special-purpose language. It requires documentation, training, tool support. In particular, you need to use a preprocessor together with a matching tool chain and compiler. Unless the preprocessor is integrated into the tool chain and shipped with every implementation, this implies lock-in and slow upgrades. It is not uncommon for the preprocessor not to work with the most current version of the compilers and tools or the underlying language. The main reason is that the preprocessor implementer doesn't get access to those compilers significantly before their own users. This commonly leads to users having to make a painful choice between using the preprocessor or the latest and greatest compiler and other tools. This creates friction between the preprocessor users and any non-preprocessor users they collaborate with. The debugging, compatibility, and portability problems persist because old compilers don't just die. It can take a large organization the better part of a decade to get everyone upgraded to the latest version (of something), just to fall behind again at the next release. For example, it took “forever” (almost a decade) to get C++ template implementations good enough for mainstream use. However, some users still rely on decade old compilers.

A preprocessed language tends to have problems interacting the type system of the host language. Having the same type system as the host language is often not good enough — after all, the purpose of a preprocessed language is to elegantly express things that cannot be expressed elegantly in the host language. Error detection and error reporting problems are just the most obvious examples of this. Concepts (a type system for types) [21, 18], as being developed to improve C++ templates’ support for generic programming and template metaprogramming, is an example of a mechanism addressing the problem of mismatch of the type systems of a higher-level language and a lower-level host language. Higher order types fills some of the same role in the specification of abstractions in functional languages.

So, a preprocessed language share many problems of with a special-purpose language with a stand-alone implementation. In fact, as their tools become more complete and their definition more precise and separate from the host language, they grow into special-purpose languages. Conversely, if their implementation and type system support becomes more integrated with the host language, they cease to be separate languages and become abstraction mechanisms of the host language (C++ templates is a prominent example). In addition, preprocessing languages tend to suffer the problems of libraries: Unless all code conforms to the conventions of the preprocessed language, the guarantees the language can rely on and offer weakens.

4.4 Dialects

Take a popular general-purpose language, add desired features to a compiler and/or a run-time support system, and you have your own private dialect. This may be the most popular way of creating a new language. The result is not quite a special-purpose language, but it has special-purpose features embedded in a general-purpose language. Working in a production-quality general-purpose language implementation is hard, though. Many people will simultaneously be making modifications in such an implementation. Furthermore, compilers, debuggers, libraries, tools are required parts of such implementations and major implementations target many platforms. Consequently, most people who extend a language in this way do so in a minor — less messy — implementation, modifying only the part of the tool chain they need, and target only the platforms they care about. This is reasonable — in many cases even essential — to allow people to focus their efforts on the design and implementation of the new facilities they want. Unfortunately, the effect is that unless the major vendors adopt the new dialect, its designers are left with a private language. This implies all the usual private language costs — and the usual mortality rate. In addition, it is essentially impossible to remove undesirable features from a dialect. Doing so would destroy compatibility and basically move the language away from the dialect classification and into the special-purpose language classification.

5 The SELL approach

The analysis in sections 3 and 4 paints a grim picture of the problems of applying language design and implementation techniques to support software development. One conclusion would be to leave the field to big corporations with deep pockets: Let them do the design, development, and apply their marketing muscle; then we live with the results, whatever they may be. An alternative conclusion is to withdraw into some cosy ghetto of our own design and let the rest of the world do what it likes without interference or input from us. I like neither alternative and point to a way to dodge the horns of this dilemma:

1. superset: Add libraries to provide application-specific facilities, then
2. subset: Subtract features (outside the library implementation) to provide semantic guarantees

The result is a subset of a superset of a language called a *Semantically Enhanced Library Language*. When subsetting we can aim at a “clean and regular” language. Since a SELL will aim for a narrower application domain than its host languages, we have a good chance of the result being simpler than its host.

We must consider this approach in terms of expressiveness (“can we really express things as well in a library as in a special-purpose language?”) and tools (“will we get stuck developing and maintaining a messy tool chain?”). The claim is that the answers can be “yes” and “no” for a large enough range of problems and a low enough cost to prefer the SELL approach over the traditional approaches mentioned in sections 3 and 4. Obviously, the SELL approach is not completely new — in fact, it is an attempt to synthesize what has worked best in the traditional approaches and dodge the worst problems. Please also note that I don’t claim that the other approaches to making special-purpose features available never work or that there are no other alternatives. That would be absurd. What I do claim is that the success rate for new languages — if measured by survival of a language for a decade and use outside the group that originated it — is very low and the costs higher than often realized.

The argument about expressiveness of libraries is based on a pair of old Bell Labs sayings:

1. Library design is language design
2. Language design is library design

We need both. In other words, the expressiveness of a library depends on the ability of a general-purpose language to define libraries. Functional programming, object-oriented programming, and generic programming are prominent schools of thought that give a prominent role to library building.

The skills needed to write a good library are very similar to the skills needed for all high-end systems programming or application building. Furthermore, when we write a library, we can rely on existing infrastructure (compilers, debuggers, libraries, education, etc.). The result is that libraries are cheap to produce compared to alternatives.

However, the tools part could easily lead us into the debugging, tool chain, and maintenance problems characteristic of dialects and preprosessors. To avoid that we need a tool for expressing constraints and high-level transformations that is minimally invasive into the tool chain. To further keep the tool problems under control, we need a general tool for doing that and one that will fit into all tool chains. That is, we need a general-purpose tool for analysing source code and performing source-level transformations that relies on a standard interface to compilers.

5.1 C++

In principle, any general-purpose programming language can be the host language for the SELL approach. Unsurprisingly, my favorite / chosen host language is C++ [19, 8]

C++ has the virtues of stretching to a very broad range of application areas, good performance, a large and lively user community, and support for compilers, libraries, and tools for essentially all platforms [22].

C++'s abstraction facilities provide adequate support for object-oriented programming, generic programming, traditional procedural programming, and multi-paradigm programing combining elements of those. Classes plus templates plus overloading is the basis of expressiveness and performance.

Obviously improvements are possible — even given the Draconian compatibility constraints imposed by the huge user community and the wide range of application areas. In particular, we hope that the next standard (C++0x) will offer concepts (a type system for types), more general and flexible facilities for initilization, and remedies for many minor annoyances [20]. Unfortunately, the compatibility constraints and the use of C++ for very low-level system components precludes remedying obvious weaknesses, such as overly agressive implicit conversions (incl. the array-to-pointer conversion) and unchecked unions.

5.2 A brief overview of the Pivot

The Pivot is a general framework for the analysis and transformation of C++ programs[13]. 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.

There are lots of (more than 20) tools for static analysis and transformation of C++ programs, e.g. [15, 2, 16, 12]. However, few — if any — handle all of ISO Standard C++ [8, 19], most are specialized to particular forms of analysis or transformation, and few will work well in combination with other tools. The design of the Pivot is focussed on advanced uses of templates as used in generic programming, template metaprogramming, and experimental use of libraries as the basis of language extension. Since (static) types is central to such libraries, the SELL approach requires 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 some language support in the next ISO C++ standard (C++0x) [21, 18, 20].

The central part of the Pivot is a fully typed abstract syntax tree called IPR (*Internal Program Representation*):

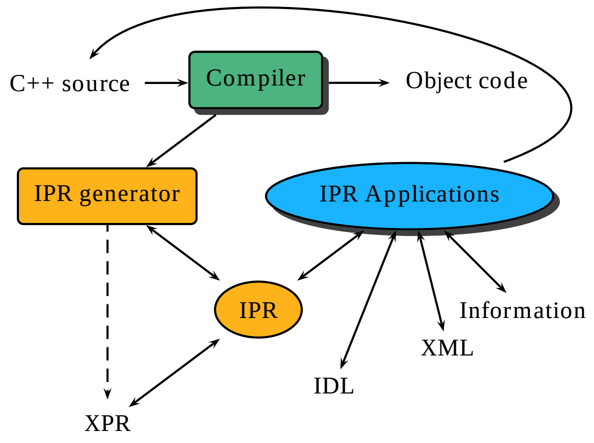

Figure 1. An overview of *The Pivot* infrastructure

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*).

The IPR is complete and arguably minimal. Traversal of C++ code represented as IPR can be done in several ways, including "ordinary graph traversal code", visitors [6], iterators [19], or tools such as Rose [15]. The needs of the application — rather than the IPR — determines what traversal method is most suitable.

Currently, the Pivot does not support an annotation language. Pivot programs can annotate IPR nodes, but there is no facility for the programmer to embed annotations in the C++ source text. Providing such a facility is easy, but once programmers starts to depend on such annotations, they have created a new special-purpose language. We want to explore how much can be done with the SELL approach, relying only on standard conforming C++ source text.

6 Examples of SELLs

The proof of the pudding is in the eating, but this is not a paper presenting you with a SELL for use; it is a presentation of the general idea of SELLs. Therefore, I present only details that will illustrate the idea of a SELL, not complete SELLs.

6.1 Safe C++

C++ inherits a host of opportunities for type violations from C and adds a few of its own. It is possible — and not very hard — to write type-safe code in C++. However, it is not easy to know that no type violations exist in a program, especially in a large program written and maintained by many programmers with a variety of backgrounds and a variety of ideas of what constitutes safe code. So, how would we support a type-safe dialect of C++ that maintains the essential expressiveness and efficiency of C++? In particular, we want to be sure that there are no type violations in the code. We can only be really sure if we can provide a tool (or combination of tools) that will detect all violations. In the absence of tools, we must rely on humans to follow rules. That would probably be better than the state of the art in most software development organizations, but it would only be second best.

Consider the major insecurities in C++ code:

1. Buffer overruns — i.e., reading or writing outsider the range of an array
2. Dereferencing an uninitialized pointer, a zero-valued pointer, or a pointer to a deleted object
3. Misuse of a union — i.e. write a union variable as one type and read it as another
4. Misuse of a cast — e.g. cast an int to a pointer type where no object of that type exist where the new pointer points
5. Misuse of `void*` — e.g. assign an `int*` to a `void*` and cast that `void*` to a `double*`
6. Deleting an object twice, not deleting an object after use, or using a pointer after deletion.

The obvious approach for avoiding these problems is to provide a library (or a set of libraries) that saves the programmer from having to use these error-prone features. For example, instead of using arrays, the programmer can use a range-checked `vector` and instead of a `union` a user can use a tagged `union` or an `Any` type. Casts (with exception of the dynamicaly type-safe `dynamic_cast`) and `void*`s are rarely useful outside low-level and easily encapsulated uses, so they can simply be avoided. If we use counted pointers, memory leaks won't happen (depending on how cyclic data structures are handled). Since pointers are checked, we don't access through invalid pointers and double deletion are easily detected.

Basically, errors that cannot be detected until run-time are systematically turned into exceptions, making *Safe C++* a dynamically type safe language. Exceptions may not be your favorite language feature, but they are useful in most contexts and are universally used for reporting run-time type violations in languages deemed type-safe.

So, we can fairly easily write code that doesn't suffer from the obvious type-safety problems. What is outlined here is a SELL where the superset is created by adding checked `vector`s, "smart" checked pointers, a tagged `union` (or an `Any` type). However, nothing has been gained if users persist using the unsafe-features in unsafe ways. For example, we can write safe code, but someone might just do something like this:

```
double* horrible(int i)
{
    int v[80];
    char* p = new char[200];
    double* q = new double[200];
    Shape* pc = new Circle(Point(10,20),20);
    delete[] p;
    p[100] = 'c';
    p[i] = 'x';
    v[100] = 666;
    pc->rotate(45);
    pc->draw();
    f(pc);
    void* vp = v;
    delete vp;
    delete[] p;
    return q;
}
```

Obviously, the subsetting (enforcement) part of the SELL design must be to detect and eliminate the unsafe uses of the host language. Please note that the tool that does that must distinguish between the use of the "banned" features or uses of features within the implementation of the extensions and direct use by the user. In this case, a dumb tool (such as a compiler option) banning all uses of pointer would prevent the use of `vector` that uses pointers internally. Instead, we could use the Pivot to catch only the uses of pointers outside our supporting classes. That done, our code would have to be rewritten to look something like:

```
unique_ptr<vector<double>> messy(int i)
{
    vector <int> v(80);
    string p(200);
    vector<double> q(200);
    scoped_ptr<Shape> pc(new Circle(Point(10,20),20));
    p[100] = 'c';       // ok
    p[i] = 'x';         // checked at run time
    v[100] = 666;       // caught at run time
    pc->rotate(45);
    pc->draw();
    f(pc);
    return unique_ptr<vector<double>>(q);
}
```

This is much better (ignoring the messy use of "magic constants"), but *Safe C++* could have problems for real-world programming in many areas where C++ is used: We have not dealt with performance and compatibility. Actually, this code hints of a very significant concern for performance in the library design: `scoped_ptr` deletes its object at the end of scope and prevents `f` from keeping a reference to that object. Similarly, `unique_ptr` cooperates with `vector` to ensure that the elements of `q` are transferred out of `messy` and not destroyed as part of `q` upon exit. We didn't just rely on counted pointers of a garbage collector to deal with resource problems.

Using the Pivot, we could do better, though. By default, both uses of `pc` in `messy` must be checked for validity (assuming that a `scoped_ptr` can be a null pointer). However, a bit of simple flow analysis can eliminate the second check, and a slighly more clever analysis will reveal that no checking is actually necessary: We can see that `pc` has not been properly initialized and not assigned to — and so can the Pivot. This kind of analysis has been used experimentally for private languages and dialects [9]. Given the Pivot, we can apply this for a library or for "raw C++".

Compatibility is a harder problem. What if `f` is not known to be safe? What if we can't rewrite or recompile all the code of a system? What if layout compatibility of some data structures is required? *Safe C++* as presented here is just an illustration, not a full-blown SELL.

6.2 Parallel C++

With the emergence of cheap multiprocessors, clusters, and multi-core chips, concurrency is increasingly important. Many languages and dialects have been designed to address the concurrency needs of high-performance scientific computing. Here I will build on a library, STAPL [1] [14], that offers parallel operations on containers in the spirit of the STL. For example:

```
void f(pvector<double>& v)
{
     prange<double> r = find_all(v.range(),criteria);
     sort(r);
     cout << r;  // ordinary serial output of elements
}
```

Imagine that `v` has 500 million elements and that the program runs on a serious supercomputer, such as Blue Gene\L[3] (where STAPL is in fact used). The `find_all` will execute in parallel on as many processors as the STAPL run-time system deems reasonable finding elements that meets `criteria`. If `find_all` finds lots of elements, then `sort` will also use many processors.

Here we have a sophisticated library combined with an even more advanced run-time support system. What can the Pivot do to help? For starters, it can produce the information that the run-time support system needs to function well. Secondly, it can provide classical flow analysis and aliasing information. Finally, it can be programmed to recognize usage patterns to allow algorithm substitution (as in the initial matrix algebra example) and alert the programmer to likely problems or opportunities.

6.3 Real-time C++

The problems of real-time code for embedded systems combine concerns for correnctness, reliability, and performance in constrained circumstances. Some problems and solutions overlap with those of *Safe C++* but others are unique in that they require that every operation is performed in a known constant time (or less). Naturally, not all real-time and embedded systems are written under this Draconian rule, but let's see how we can address those that do. Some C++ operatons become unusable:

1. free store (general `new` and `delete`)
2. exceptions (assuming inability to easily predict the cost of a `throw`)
3. class hierarchy navigation (`dynamic_cast` in the absence of a constant time implementation [7])

First, we add a suitable support library:

1. a fixed size `Array` class (no conversion to pointer, knows its own size)
2. some safe pointer classes
3. memory allocation classes that guarantee constant time allocation (and deallocation if allowed) — pools, stacks, etc.
4. ...

Next, we use the Pivot to eliminate dangerous operations (as listed in §6.1) from user code.

In principle, this will do the job. However, we can do more. For most programs of this sort, we can do whole-program analysis. Such programs tend to be relatively small and not allow dynamic linking. Thus, the Pivot could be used to allow exceptions for error reporting: we can verify that every exception is caught and calculate the upper bound for each throw. This is a special — and espicially hard — example of using a tool to verify that resource consumption is within acceptable bounds.

In general, there are lots more that the Pivot can do in the context of embedded systems. Some depends on a specific application, so the boundary between SELL and application support blurrs. For example, it is not uncommon for an embedded program to be more permissive about the facilities that can be used during a startup phase. The SELL can define what "startup" means (e.g. called from `start_up`) and only apply the stringent rules outside that.

7 Conclusions

The first half of this paper outlines the problems facing programmers providing and using a special-purpose language defined in the most common ways: as a separate language, as compiler options, as libraries, using a pre-processor for a general-purpose language, and as a dialect. The picture painted is bleak, leading to a suggested alternative: *Semantically Enhanced Library Languages* (SELLs) The SELL approach offers a practical and economical alternative to the more common ways of implementing extensions, dialects, and special-purpose languages. By using libraries, it limits the problems with compatibility and tool chains. By adding tool support, it enhances the appeal of libraries.

8 Acknowlegements

Thanks to Gabriel Dos Reis and my students for reading drafts of this paper and making many detailed suggestions. Also thanks to the anonymous reviewers for making more major suggestions, some of which will require several more papers.

9 References

[1] Ping An, Alin Jula, Silvius Rus, Steven Saunders, Tim Smith, Gabriel Tanase, Nathan Thomas, Nancy Amato, Lawrence Rauchwerger: *STAPL: An Adaptive, Generic Parallel C++ Library* In Wkshp. on Lang. and Comp. for Par. Comp. (LCPC), pp. 193-208, Cumberland Falls, Kentucky, Aug 2001.

[2] 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/`.

[3] IBM: `http://www.research.ibm.com/bluegene/`.

[4] William R. Mark, et al: *Cg: A System for Programming Graphics Hardware in a C-like Language*. Proceedings of SIGGRAPH 2003.

[5] M. Fernandez, et al: *SilkRoute: A framework for publishing relational data in XML*. ACM Trans. Database Syst. 27(4): 438-493 (2002)

[6] Erich Gamma, et al: *Design Patterns*. Addison-Wesley, 1994.

[7] Michael Gibbs and Bjarne Stroustrup: *Fast Dynamic Casting*. Software—Practice & Experience. Vol 35, Issue 686. 2005.

[8] International Organization for Standards, *International Standard ISO/IEC 14882. Programming Languages — C++*, 2nd ed., 2003. Wiley 2003. ISBN 0-470-84674-7.

[9] Trevor Jim, et al: *Cyclone: A Safe Dialect of C*. USENIX Annual Technical Conference, pages 275–288, Monterey, CA, June 2002.

[10] Lengauer, et al: *Domain-specific program generation*. Revised papers deom Dagstuhl seminar. March 2003. LNCS 3016.

[11] Diane J. Litman, Anil K. Mishra, and Peter F. Patel-Schneider: *Modeling Dynamic Collections of Interdependent Objects Using Path-Based Rules*. Proc. 12th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA-97). October 1997. `http://www.research.att.com/sw/tools/r++/` and `http://www.bell-labs.com/project/r++/`.

[12] George C. Necula, et al: *CIL: Intermediate Language and Tools for Analysis and Transformation*. `http://manju.cs.berkeley.edu/cil/`.

[13] The pivot is a program analysis and transformation infrastructure being developed at Texas A&M University.

[14] Steven Saunders, Lawrence Rauchwerger: *ARMI: An Adaptive, Platform Independent Communication Library* In Proc. ACM SIGPLAN Symp. Prin. Prac. Par. Prog. (PPOPP), pp. 12, San Diego, CA, Jun 2003. `http://parasol.tamu.edu/groups/rwergergroup/research/stapl/`.

[15] 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).

[16] 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).

[17] Jeremy G. Siek Andrew Lumsdaine: *The Matrix Template Library: A Generic Programming Approach to High Performance Numerical Linear Algebra*. ISCOPE'98, vol. 1505 of Lecture Notes in Computer Science, 1998.

[18] 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.

[19] B. Stroustrup, *The C++ Programming Language*, special ed., Addison-Wesley, 2000. ISBN 0-201-70073-5 .

[20] B. Stroustrup: *The design of C++0x*. The C/C++ Users Journal. May 2005.

[21] B. Stroustrup, G. Dos Reis: *A concept design*. Technical Report N1782=05-0042, ISO/IEC SC22/JTC1/WG21, April 2005.

[22] B. Stroustrup: *Examples of C++ applications*: `http://www.research.att.com/~bs/applications.html`. *Some C++ compilers*: `http://www.research.att.com/~bs/compilers.html`.

[23] Tod Veldhiusen: *Arrays in Blitz++* ISCOPE'98, vol. 1505 of Lecture Notes in Computer Science, 1998.

[24] Wilson and Lu (editors): *Parallel programming using C++*. Addison-Wesley. 1996. ISBN 0-262-73118-5.