ABI Compatibility Through a Customizable Language

Kevin Atkinson Matthew Flatt Gary Lindstrom
University of Utah, School of Computing
{kevina,mflatt,gary}@cs.utah.edu

Abstract

ZL is a C++-compatible language in which high-level constructs, such as classes, are defined using macros over a C-like core language. This approach makes many parts of the language easily customizable. For example, since the class construct can be defined using macros, a programmer can have complete control over the memory layout of objects. Using this capability, a programmer can mitigate certain problems in software evolution such as fragile ABIs (Application Binary Interfaces) due to software changes and incompatible ABIs due to compiler changes. In this paper, we outline the problem of fragile and incompatible ABIs and show how ZL can be used to solve them.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Compilers

General Terms Languages, Design

Keywords Macros, C, C++, ABI, Binary Compatibility

1. Introduction

There are two types of programming interfaces to a library: the *Application Programming Interface* (API) and the *Application Binary Interface* (ABI). The API defines the ways a programmer may request services from the library. Some of the constituents of an API in an object-oriented language are the names of classes, the methods they support, and the types of the arguments that methods take. What goes into the API is under the control of the library designer. An ABI is the object-code equivalent of an API. It is the low-level interface between the application and the library. A compiler implements a mapping from a library's API to its ABI. Some of the constituents of the mapping include calling conventions and class layout. Unlike the API, the programmer has little to no control of the ABI in most languages.

When a library designer changes an API in a way that preserves backwards compatibility with previous releases, *source code compatibility* is maintained. That is, existing applications that use a library do not need to change at the source level. However, even if source code compatibility is preserved, *binary compatibility* need not be preserved; existing applications may need to be recompiled because the compiler typically does not guarantee ABI compatibility with API compatibility.

In situations when a library is used by a small number of programs that can easily be recompiled, breaking binary compatibility between releases may be acceptable. However, if a large number of programs depend on the library, then recompiling is not an acceptable option as it can take anywhere from hours to days to recompile everything. In addition, in many situations the source code for applications using the library is not available, thus making upgrading impossible unless binary compatibility is preserved.

Preserving binary compatibility for C++ programs is difficult because the typical C++ ABI is extremely fragile. Seemingly simple changes, such as adding methods, may break binary compatibility. In fact, almost any change to a class declaration will likely break binary compatibility and require applications that use the library to be recompiled. This is a major problem in C++ software evolution, especially because the source code of an application is not always available when libraries need to be upgraded.

In addition, the C++ ABI is not well defined as every compiler implements the C++ standard in a slightly different way. Libraries compiled with one compiler, such as Visual C++, generally will not be usable by applications compiled with a different compiler, such as GCC. Furthermore, the ABI may change between releases of the same compiler. Thus, upgrading to a newer compiler may also break binary compatibility.

In contrast to C++, the C ABI is simple and well defined for a given architecture and operating system. Since the C ABI is far simpler than the C++ ABI, preserving binary compatibility is much easier. Furthermore, since the C ABI is well defined for a given architecture, compatibility between compilers is a non-issue. In fact, some C++ applications export only a C ABI for these very reasons.

The C ABI is successful because of its simplicity and consistency. That simplicity, in turn, is based in part on the simplicity of the C language. As languages become more complicated, so do the number of choices to be made in an ABI. Thus, ABIs for complicated languages, such as C++, tend to vary among compilers and even among versions of a compiler. Standardizing on one C++ ABI would solve the incompatibility problem. Although some effort has been made in that area with Itanium C++ ABI [1], there are still several C++ ABIs in common use, most notably the GCC and Visual C++ ABIs.

Even if all C++ compilers standardized on a single ABI, the problem of preserving binary compatibility between releases of a library would still be a major problem. This is because most C++ ABIs, including the Itanium C++ ABI, are optimized for performance, not preserving binary compatibility. Previous designs for a less fragile ABI for C++ [2, 3] make significant sacrifices in performance. Thus, library designers must make a choice between breaking binary compatibility between releases or contorting their programs to preserve it by using a variety of programming idioms.

We could try to add a few extensions to C++, such as a choice of different ABIs or support for common programming idioms, but a fixed number of extensions will never be enough as the problem of preserving binary compatibility is far too complex. A monolithic language cannot and should not support every possible

The definitive version was published in *Proceedings of the Ninth International Conference on Generative Programming and Component Engineering (GPCE'10)*, October 10–13, 2010, Eindhoven, The Netherlands. http://dx.doi.org/10.1145/1868294.1868316

rarely needed case. A more general and integrated approach is an extensible compiler. Traditional extensible compiler designs treat a compiler extension as an entity separate from the code to be compiled. On the other hand, a *macro system* acts as an extensible compiler and also allows the programmer to implement code and compiler extensions together, thus elevating compiler extensions to the level of a library. This, in turn, allows different ABI choices to be incorporated with different parts of an application. For example, one class can use an ABI optimized for performance while another uses an ABI aimed at preserving binary compatibility.

A simple macro system, such as the C preprocessor, is not adequate for defining compiler extensions. Rather, the macro system must be an integral part of the language that can do more than rearrange syntax. In addition to providing macro primitives, a language for giving the programmer control over an ABI must include a carefully designed core that allows higher-level constructs, such as classes, to be implemented via macros. This capacity enables the programmer to redefine key aspects that affect ABI attributes, such as class layout.

ZL, our new C++ compatible systems programming language in development, does exactly this. Our contribution in this paper is to demonstrate how ZL can be used to mitigate the problem of binary incompatibility through the use of macros. For example, we show how to avoid breaking binary compatibility when adding new data members or methods to a class. We also demonstrate how to match other compilers' ABIs with alternative class implementations, and how classes with different ABIs can be used in the same program.

The rest of this paper is organized as follows. Section 2 explores in detail the the fragility of C++ ABIs and how a macro language can help. Section 3 describes ZL. Section 4 illustrates how ZL can be used mitigate some of the example problems from Section 2. Section 5 summarizes the current state of the ZL compiler.

2. The C++ ABI and Binary Compatibility

There are many components to a C++ ABI. The primary components are: the parts in common with C, such as function calling conventions; class layout; name mangling, which includes naming of template instances; linkage specification; and exceptions. This paper focuses primarily on class layout since it causes the most problems with binary comparability, but our solution is designed to apply to the other problems as well.

2.1 Binary Compatibility

One way to break binary compatibility in C++ is to add new data members to a class. This change breaks binary compatibility because it changes the size of the class, which is used at compile time when allocating objects on a stack or inlining one object in another. A solution to this problem is to fix the size of a class by adding dummy members to allow room for future expansion. That way, the class size can remain constant. Another solution is to store all private data members in a separate object, commonly known as the pimpl (short for Private Implementation) idiom. A hybrid approach, which can be used when the class size is already fixed but there is not enough room for all the new data members, is to store only the new data members in a separate object.

All of these idioms are possible to implement without any additional support in C++. However, the first solution, adding dummy data members, is non-portable because it depends on knowing the size of the object on a particular architecture. With some clever programming it possible to avoid having to know those sizes, but none of the solutions is particularly elegant or transparent. The pimpl idiom is tedious for the programmer to use as all accesses to private data members require an extra level of indirection. The hybrid approach is even worse as some of the data members are accessed directly and others require indirection. Furthermore, switching techniques requires changing all methods that use the private data member.

Just as adding data members can break binary compatibility, so can adding new virtual methods. Adding virtual methods changes the size of the vtable, and thus, with most ABIs, changes the offsets of all the methods' function pointers for any subclasses. However, unlike adding new data members to a class, the C++ programmer's options are limited. It is possible to fix the size of the vtable by adding dummy methods, but techniques such as the pimpl idiom do not apply since the programmer has no control over how the vtable is laid out.

Reordering methods in a class declaration can break binary compatibility, because it changes the offsets of the methods' function pointers in the vtable. There is no real solution to this in C++, other than to just be aware of this fact and not do it.

Removing data members and methods will break ABI compatibility in C++, because it changes the offset of any members after the removed one. The only way to solve this problem in C++ is to avoid removing the member, or by instead replacing it with a dummy member. This way the layout is preserved. The unused slot can later be replaced with a new member in order to save space, or it can simply be left unused.

Adding parameters to a function or method in C++ also breaks binary compatibility,[1] because it changes the mangled name of the symbol used to represent the function. This occurs because the types of the parameters are encoded as part of the symbol to support overloading. Since C++ allows for overloading, a programmer can avoid this problem by defining a new function with the added parameter. The old function can then call the new one.

Changing compilers can also break binary compatibility, since ABIs differ between compilers and sometimes between different versions of the same compiler. Thus, when using C++ libraries, not only is the specific version of the library important, but so is the compiler used to compile it. Unfortunately, there is no good solution to this problem in C++, other than always using a compatible compiler when compiling the library. The only way to support a different, incompatible, compiler is to avoid directly using the C++ ABI altogether. A typical work-around is to create a C API on top of the internal C++ API, and then only export the C API. This technique effectively defines a program-specific ABI that the library developer has complete control over. It may seem silly for a C++ program to have to use a C API to use another C++ library, but currently there is no other way around the problem.

2.2 Macros to the Rescue!

A good macro system can automate all the programming idioms of the previous section. In addition, when programming idioms are not sufficient, a macro system that defines classes via macros gives the library implementer the ability to devise novel ways to control the class layout. The implementers can either replace the existing macro implementation, or, if class macro is designed to be customizable, tweak the existing ABI by reusing the existing implementation.

Here are some of the many ways a macro system can help:

- *Adding new data members.* When fixing the size of a class, a macro can manage the size of the object and not the programmer. When implementing the pimpl idiom through a macro, the extra indirection is invisible to the programmer. Furthermore, the programmer can switch from one technique to another without large-scale source code changes.

[1] Since C++ supports default values for parameters, adding arguments does not necessarily break source-code compatibility.

- *Adding new virtual methods.* With a macro system that implements vtables as classes, all of the techniques that apply to classes also apply to the vtable. All that is needed is a way to specify how the vtable class is implemented, something that is trivial if classes are implemented via macros.
- *Reordering methods.* With a macro system it is possible to avoid the problem by storing the offsets in a separate interface file that will be updated when new methods are added. When combined with fixing the size of the vtable, this technique will also allow methods to be added anywhere in the class declaration, rather than having to add them at the end.
- *Removing data members.* A macro system can help by freezing the layout as just discussed. When this is done, the macro system can automatically insert a dummy member in place of the removed member. Later on, the unused slot can be replaced with a new member.
- *Matching existing ABIs.* If classes are implemented via macros, then the programmer has control of how classes are implemented, and thus has control over which ABI is used. In fact, a programmer can use classes with different ABIs within the same program. For example, the ABI used can be specified as part of the class declaration. For using existing code, the ABI can be specified on a per header-file basis.

The above are some of the more common solutions to ABI problems. None of the solutions is perfect. For example, of the techniques used to preserve binary compatibility when adding new data members, reserving space ahead of time is a good solution when performance really matters but requires planning ahead; the pimpl idiom has a small performance overhead which some may find unacceptable; and the hybrid approach could be seen as overly complex. In addition there are many more, less common techniques for preserving binary compatibility not listed here. Thus, due to the sheer number of solutions and the various trade-offs involved, none of them are good candidates for language extensions. However, with a good macro system, the solutions can be implemented as libraries.

2.3 User Roles

A good macro system can benefit all users, but not everyone needs to know the full details of how macros work. There are three primary classes of users: 1) *End Users or Library Consumers*, who just use the library, but can benefit from increased binary compatibility; 2) *Library Implementers*, who can use the macro libraries to provide increased binary comparability, but do not need to know the details of the macro libraries themselves; and 3) *Tool Implementers*, who provide the macro libraries for the library implementers.

With traditional compiler designs, tool implementers are in relatively short supply, and they face a daunting task on two fronts: they must modify the compiler, and they must convince users of the library to use the modified compiler. Our approach to improving ABI compatibility is to simplify the tool implementer's job, so that library implementers will have better tools and end users will have more compatible libraries. Specifically, with a macro-extensible compiler that can express ABI details through the macro layer, tool implementers gain a simpler framework for implementing more interoperable designs, and they get a more composable framework so that multiple tools can be combined. In this way, a tool becomes more like a library.

Indeed, just as library consumers can become library implementers when they want to generalize their application code so that others can use it, library implementers can become tool implementers when they need to do something unusual for which a macro library does not yet exist. The key benefit of a macro system in this case is that it allows a library implementer to easily become a tool implementer.

3. ZL

ZL is a C++-compatible language that solves ABI compatibility problems by giving the programmer as much control as possible. ZL provides a C-like core and enough of C++ to let the type-checker and compiler do its job without committing to key parts of the ABI such as class layout. The rest is defined using a sophisticated macro system.

The ZL library provides a default implementation of language constructs such as classes. The implementation can be overridden or extended by defining new macros in a source file or by importing a macro library. Macros, including those that define the behavior of a language construct, are scoped and can be shadowed. This means it is possible to use two different class ABIs by loading one class library and defining some classes, then loading another library and define some more classes. A more convenient solution is to add some syntax for selecting the ABI for a class, which ZL also supports.

ZL provides two kind of macros: *pattern-based macros* that simply rearrange syntax, and *procedural macros* that are functions that perform more complex manipulation of syntax or take action based on the input, as is necessary to implement classes. In addition ZL provides user types which are the building blocks for classes.

3.1 Macros

The simplest form of a macro is a *pattern-based macro*, which is simply a transformation of one piece of syntax to another. For example, consider an `or` macro that behaves like C's `||` operator, but instead of returning true or false, returns the first non-zero value. Thus, `or(0.0, 6.8)` returns `6.8`. To define it, one uses ZL's `macro` form, which declares a pattern-based macro:

```
macro or(x, y) { ({typeof(x) t = x; t ? t : y;}); }
```

In ZL, as in GCC, the `({...})` is a statement expression whose value is the result of the last expression, and `typeof(x)` gets the type of a variable. Like Scheme macros [4], ZL macros are hygienic, which means that they respect lexical scope. For example, the `t` used in `or(0.0, t)` and the `t` introduced by the `or` macro remain separate, even though they have the same symbol name.

The `or` macro above has two *positional* parameters. Macros can also have *keyword* parameters and *default values*. For example:

```
macro sort(list, :compar = strcmp) {...}
```

defines the macro `sort`, which takes the keyword argument `compar`, with a default value of `strcmp`. A call to sort will look something like `sort(list, :compar = mycmp)`.

3.2 Classes and User Types

Most of the class implementation in ZL is left to macros, but since classes are an integral part of the C++ type system, ZL still needs to have some notion of what a class is. *User types* are ZL's minimal notion of classes. A user type has two parts: a type, generally a `struct`, to hold the data for the class instance, and a collection of symbols for manipulating the data.

The collection of symbols is a *module*. For example

```
module M { int x;
           int foo(); }
```

defines a module with two symbols. Module symbols are used by either importing them into the current namespace, or by using the special syntax `M::x`, which accesses the `x` variable in the above module.

A user type is created by using the `user_type` primitive, which serves as the module associated with the user type. A type for the instance data is specified using `associate_type`.

As an example, the class[2]

```
class C { int i;
          int f(int j) {return i + j;} };
```

expands to something like:

```
user_type C {
  struct Data {int i;};
  associate_type struct Data;
  macro i (:this ths = this) {(*(C *)ths)..i;}
  macro f(j, :this ths = this) {f‘internal(ths, j);}
  int f‘internal(C * fluid this, int j) {return i + j;}
}
```

which creates a user type `C` to represent a class `C`; the structural type `Data` is used for the underlying storage.

To allow user types to behave like classes, member-access syntax gets special treatment. For example, if `x` is an instance of the user type above, `x.i` calls the `i` macro in the `C` module, and it passes a pointer to `x` as the `this` keyword argument. This protocol allows `x.i` to expand to something that accesses the `x` field of the underlying struct, which can be done using the special syntax `x..i`. Thus, `i` effectively becomes a data member of `x`. Methods can similarly be defined. For example, `x.f(12)` will call the `f` macro with one positional parameter and the `this` keyword argument.

The default value for the `this` keyword argument is necessary to support the implicit `this` variable when data members and methods are accessed inside method definitions. The function `f‘internal`, which implements the `f` method, demonstrates this. (The `‘internal` simply specifies an alternative namespace for the `f` symbol so that it does not conflict with the `f` macro.) The first parameter of the function is `this`, which puts the symbol into the local environment. When `i` is called inside the function body the `this` keyword argument is not supplied, since we are not using the member access form. Therefore, the `this` keyword argument defaults to the `this` specified as the default value, which binds to the `this` in the local environment. The `fluid` keyword is necessary to makes the `this` variable visible to the `i` macro; with normal hygiene rules, binding forms at the call site of a macro are invisible, as symbols normally bind to whatever is visible where the macro was defined.

User types can also be declared to have a subtype relationship. The declaration specifies a macro for performing both casts to and from the subtype. Subtypes are used to implement inheritance. For example the class:

```
class D : public C { int j; };
```

expands to something like:

```
user_type D {
  import C;
  struct Data {struct C::Data parent; int j;};
  associate_type struct Data;
  macro _up_cast (ths) {&(*ths)..parent;}
  macro _down_cast (other) {(D*)other;}
  make_subtype C _up_cast _down_cast;
  macro j (:this ths = this) {(*(D*)ths)..j;}
}
```

New symbols defined in a module are allowed to shadow imported symbols, so the fact that there is also a `Data` in `C` does not create a problem. Also, note that there is no need to redefine the data member and method macros imported from `C`, since the existing ones will work just fine. They work because the class macro makes sure that the `ths` macro parameter is cast to the right type before anything is done with it. For example, if `y` is an instance of the type `D`, then `y.i` expands to `(*(C*)&y)..i`. When ZL tries to cast `&y` to `C*`, the `D::_up_cast` macro is called and the expression expands to `((*&(*&y)..parent))..i`, which simplifies to `y..parent..i`. Method calls expand similarly, except that the cast is implicit when the `ths` macro parameter is passed into the function.

If a class contains any virtual methods, then a vtable is also created. The macro that implements the method then looks up the function in the vtable instead of calling it directly. For example, if `f` was a virtual function in the class `C`, then the macro for `f` would look something like:

```
macro f(j, :this ths = this) {_vptr->f(ths, j);}
```

where `_vptr` is a hidden member of the class that contains a pointer to the virtual table. The vtable is also a class, so to implement inheritance with virtual methods a child's vtable simply inherits the vtable of the parent. To override a method, the constructor for the child's vtable simply assigns a new value to the entry for the method's function pointer.

3.3 Parsing and Expanding

The macros shown so far are pattern-based macros. Writing more sophisticated procedural macros, such as those required to implement classes, requires some knowledge of parsing and macro expansion in ZL. This section gives the necessary background material, while the next section details how to write such macros.

To deal with C's idiosyncratic syntax while also allowing the syntax to be extensible, ZL does not parse a program in a single pass. Instead, it uses an iterative-deepening approach to parsing. The program is first separated into a list of partly parsed declarations. Each declaration is then parsed. As it is being parsed and macros are expanded, sub-parts, such as code between grouping characters, are further separated.

ZL's iterative-deepening strategy is needed because ZL does not initially know how to parse any part of the syntax involved with a macro. When ZL encounters something that looks like a function call, such as `f(x + 2, y)`, it does not know if it is a true function call or a macro use. If it is a macro use, the arguments could be expressions, statements, or arbitrary syntax fragments, depending on the context in which they appear in the expansion. Similarly, ZL cannot directly parse the body of a macro declaration, as it does not know the context in which the macro ultimately will be used.

More precisely, the ZL parsing process involves three intertwined phases. In the first phase *raw text*, such as `(x+2)`, is parsed. Raw text is converted into an intermediate form known as a *syntax object*, but which can still have raw-text components. (Throughout this paper we show syntax objects as S-expressions, such as `("()" "x+2")`.) In the second phase, the syntax object is expanded as necessary and transformed into other syntax objects by expanding macros until a fixed point is reached. In the third phase, the fully expanded syntax object is compiled into an *AST*, which is shown as a rounded block in Figure 1.

Figure 1 demonstrates ZL's parsing and expansion process. The top box contains a simple program as raw text, which is first parsed. The result is a *syntax list* (internally represent as a `@`) of `stmt`'s where each `stmt` is essentially a list of tokens, as shown in the second box. Each statement is then expanded and compiled in turn, and is added to the top-level environment (which can be thought of as an AST node). The third box in the figure shows how this is done, which requires recursive parsing and expansion. The first `stmt` is compiled into the `fun f`, while the body of the function is left unparsed. Next, the `fun` is compiled into an AST. During the compilation, the body is expanded. Since it is raw text, this process

[2] For simplicity, we leave off access control declarations and assume all members are public in this paper.

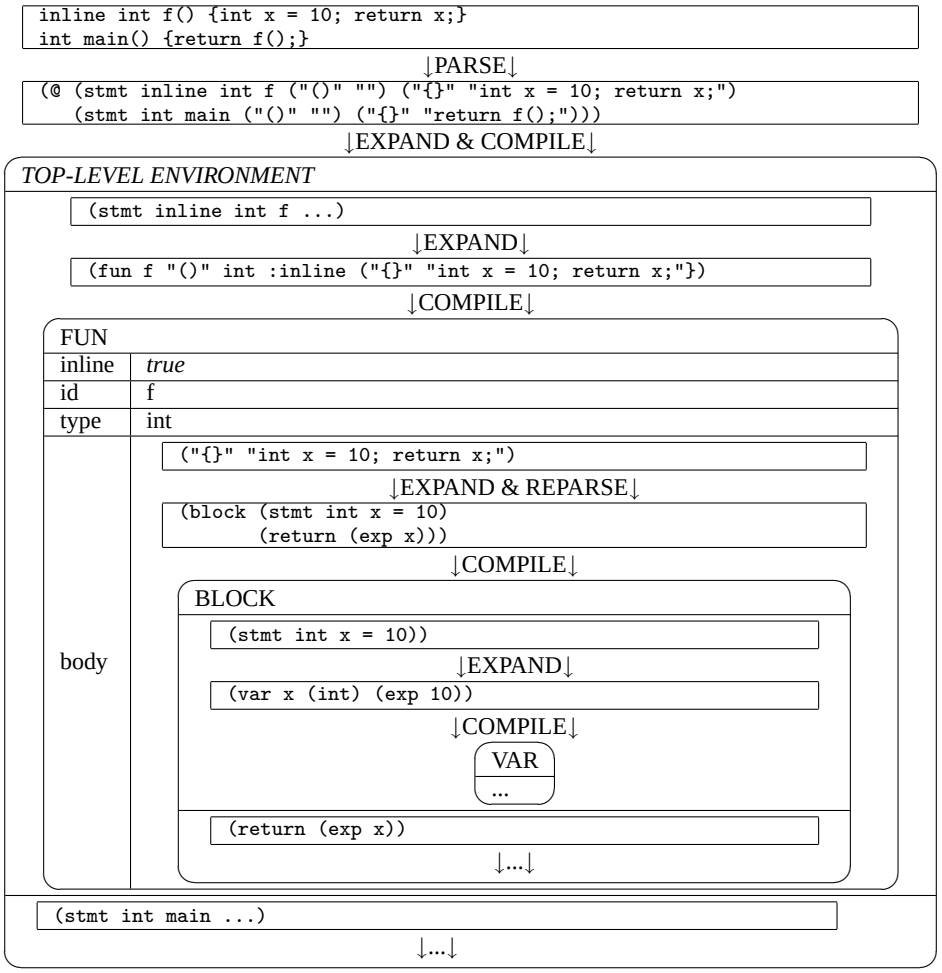

Figure 1. How ZL compiles a simple program. The body of `f` is reparsed and expanded as it is being compiled.

involves parsing it further, which results in a `block`. Parsing the block involves expanding and compiling the sub-parts. Eventually, all of the sub-parts are expanded and compiled, and the fully parsed AST is added to the top-level environment. This process is repeated for the function `main`, after which the program is fully compiled.

3.4 Procedural Macros

Some macros must take action based on the input. One example is the built-in class macro. Another example is a macro that fixes the size of the class, since the amount of padding it needs to add depends on the numeric value of the size passed in. For these situations, ZL provides *procedural macros*, which are functions that transform syntax objects.

Figure 2 demonstrates the essential parts of any procedural macro. The macro is defined as a function that takes a syntax object and environment, and returns a transformed syntax object. Syntax is created using the `syntax` form. The `match` function is used to decompose the input while the `replace` function is used to rebuild the output. Finally, `make_macro` is used to create a macro from a function. More interesting macros use additional API functions to take action based on the input. Figure 3 defines the key parts of the macro API, which we describe in the rest of this section.

```
Syntax * or(Syntax * p, Environ *) {
  Match * m = match(NULL, syntax (_, x, y), p);
  return replace(syntax
                   {({typeof(x) t = x; t ? t : y;});},
                 m, new_mark());
}
make_macro or;
```

Figure 2. Procedural macro version of `or` macro from Section 3.1

Syntax is created using the `syntax` and `raw_syntax` forms. The different forms create different types of code fragments. In most cases, the `syntax {...}` form can be used, such as when a code fragment is part of the resulting expansion; the braces will not be in the resulting syntax. If an explicit list is needed, for example, when passed to `match` as in Figure 2, then the `syntax (...)` form should be used. Neither of these forms create syntax directly, however. For example `syntax {x + y;}` is first parsed as `("{}" "x + y;")`, before eventually becoming `(plus x y)`. When it is necessary to create syntax directly, the `syntax ID` form can be used for simple identifiers; for more complicated fragments the `raw_syntax` form can be used in which the syntax is given in S-expression form.

Syntax forms:

```
new_mark() — returns Mark *
syntax (...)|{...}|ID — returns UnmarkedSyntax *
raw_syntax (...) — returns UnmarkedSyntax *
make_macro ID [ID];
```

Call back functions:

```
Match * match(Match * prev, UnmarkedSyntax * pattern,
              Syntax * with)
Match * match_args(Match *, UnmarkedSyntax * pattern,
              Syntax * with)
Syntax * match_var(Match *, UnmarkedSyntax * var);
Syntax * replace(UnmarkedSyntax *, Match *, Mark *)
size_t ct_value(Syntax *, Environ *)
Syntax * error(Syntax *, const char *, ...)
```

Figure 3. Macro API

The `match` function decomposes the input. It matches pattern variables (the second parameter) with the arguments of the macro (the third parameter). If it is successful, it prepends the results to `prev` (the first parameter) and returns the new list. If `prev` is `NULL`, then it is treated as an empty list. In the match pattern a `_` can be used to mean "don't care." The match is done from the first part of the syntax object. That is, given `(plus x y)`, the first match is `plus`. Since the first part is generally not relevant, ZL provides `match_args`, which is like `match` except that the first part is ignored. For example, `match_args` could have been used instead of `match` in Figure 2.

The `replace` function is used to rebuild the output. It takes a syntax object (the first parameter, and generally created with `syntax`), replaces the pattern variables inside it with the values stored in the `Match` object (the second parameter), and returns a new `Syntax` object.

The final argument to `replace` is the *mark*, which is used to implement hygiene. A mark captures the lexical context in which the macro is defined. Syntax objects created with `syntax` do not have any lexical information associated with them, and are thus *unmarked* (represented with the type `UnmarkedSyntax`). Therefore, it is necessary to attach lexical information to them by passing in a mark created with `new_mark` (the third parameter to `replace`).

Match variables exist only inside the `Match` object. When it is necessary to access them directly, for example, to get a compile-time value, `match_var` can be used; it returns the variable as a `Syntax` object, or `NULL` if the match variable does not exist. If the compile-time value of a syntax object is needed, `ct_value` can be used, which will expand and parse the syntax object and return the value as an integer. When it is necessary to report an error message, the `error` function can be used; it creates a syntax object that results in an error when parsed.

Once the function for a procedural macro is defined, it must be declared as a macro using `make_macro`.

This section only gives a small part of the macro API. Some of the more important functions not shown here include functions for controlling the visibility of macros and partly expanding syntax.

3.5 The Class Macro

We have now presented most of the necessary parts that make up the class macro. Sections 3.1 and 3.2 give a good idea of the code generated, while Sections 3.3 and 3.4 give a good idea of what is necessary to generate that code. However, the class macro uses more of ZL than we have presented here, which includes more of ZL's macro API. It also uses ZL's support for *syntax macros*, which work with arbitrary syntax, as opposed to *function-call macros*, which only work with syntax that takes the shape of a function call or identifier.

The core class macro is currently around 900 lines of code. The implementation is highly reusable, because it is a class itself that is organized around methods that can be overridden to extend its functionality. The bootstrapping problem of writing methods to implement classes is solved by having a simpler, more compact class system just to implement the class macro.

In addition to overriding individual methods, the `class` syntax object can be declared to expand to a completely different macro. The class macro is defined using the function `parse_class`, which can be called directly so that the new macro can reuse the original implementation.

4. Mitigating ABI Problems with ZL

ZL can be used to mitigate many of the ABI problems discussed in Section 2. This section will give the details.

4.1 Adding Data Members without Changing Class Size

Adding data members to a class changes the size of the class, which will break binary compatibility. To avoid this we must somehow fix the size of the class.

Fixing the Size of a Class. As described in Section 2, one common technique to fix the size of the class is to add dummy data members as placeholders to allow for future expansion. Using the ZL macro system, it is possible to automate this solution, as shown in Figure 4. To support this extension the ZL grammar has been enhanced to support specifying the size. The syntax for the new class form is:

```
class C : fix_size(20) {...};
```

which will allow a macro to fix the size of the class `C` to 20 bytes.

The macro in Figure 4 redefines the built in `class` macro. It works by parsing the class declaration and taking its size. If the size is smaller than the required size, an array of characters is added to the end of the class to make it the required size.

The details are as follows. Lines 2–7 decompose the class syntax object in order to extract the relevant parts of the class we need. The `@` by itself means that anything after it is optional. The `pattern` form matches the sub-parts of a syntax object. The `@` before a identifier means to match any remaining parameters and store them in a syntax list; thus, `body` contains a list of the declarations for the class. Here, `:(fix_size fix_size)` means to match a keyword argument of a syntax object; the first part is the syntax to match against, and the second is the pattern variable to store the results in.

If the class does not have a body (i.e. a forward declaration) or a declared `fix_size`, then the class is passed on to the original class macro in line 9. Line 11 compiles the `fix_size` syntax object to get an integer value.

Lines 13–21 involve finding the original size of the class. Due to alignment issues the `sizeof` operator cannot be used, since a class such as "`class D {int x; char c;}`" has a packed size of 5 on most 32 bit architectures, but `sizeof(D)` will return 8. Thus, to get the packed size a dummy member is added to the class. For example, the class `D` will become "`class D {int x; char c; char dummy;}`" and then the offset of the dummy member with respect to the class `D` is taken. This new class is created in lines 13–17. Here, the `@` before the identifier in the replacement template splices in the values of the syntax list.

To take the offset of the dummy member of the temporary class, it is necessary to parse the class and get it into an environment. However, we do not want to affect the outside environment with the temporary class. Thus, a new temporary environment is created

```
1  Syntax * parse_myclass(Syntax * p, Environ * env) {
2    Mark * mark = new_mark();
3    Match * m = match_args(0, raw_syntax
4        (name @ (pattern ({...} @body))
5          :(fix_size fix_size) @rest), p);
6    Syntax * body = match_var(m, syntax body);
7    Syntax * fix_size_s = match_var(m, syntax fix_size);
8
9    if (!body || !fix_size_s) return parse_class(p, env);
10
11   size_t fix_size = ct_value(fix_size_s, env);
12
13   m = match(m, syntax dummy_decl,
14             replace(syntax {char dummy;}, NULL, mark));
15   Syntax * tmp_class = replace(raw_syntax
16       (class name ({...} @body dummy_decl) @rest),
17                         m, mark);
18   Environ * lenv = temp_environ(env);
19   pre_parse(tmp_class, lenv);
20   size_t size = ct_value(replace(syntax
21       (offsetof(name, dummy)), m, mark), lenv);
22
23   if (size == fix_size)
24     return replace(raw_syntax
25         (class name ({...} @body) @rest), m, mark);
26   else if (size < fix_size) {
27     char buf[32];
28     snprintf(buf, 32, "{char d[%u];}", fix_size-size);
29     m = match(m, syntax buf,
30             replace(string_to_syntax(buf), NULL, mark));
31     return replace(raw_syntax
32         (class name ({...} @body buf) @rest), m, mark);}
33   else
34     return error(p,"Size of class larger than fix_size");
35 }
36 make_syntax_macro class parse_myclass;
```

Figure 4. Expanding classes to support fixing the size of the class

in line 18. The `temp_environ` macro API function does this by creating a new scope. Line 19 then parses the new class and adds it to the temporary environment. The `pre_parse` API function partly expands the passed-in syntax object and then parses just enough of the result to get basic information about symbols.

With the temporary class now parsed, lines 20–21 get the size of the class using the `offsetof` primitive.

Lines 23–34 then act based on the size of the class. If the size is the same as the desired size, there is nothing to do and the class is reconstructed without the `fix_size` property (lines 23–25). If the class size is smaller than the desired size, then the class is reconstructed with an array of characters at the end to get the desired size (lines 26–32). (The `string_to_syntax` API function simply converts a string to a syntax object.) Finally, an error is returned if the class size is larger than the desired size (lines 33–34).

The last line declares the function `parse_myclass` as a syntax macro for the `class` syntax form.

Allowing Expansion. The example in Figure 4 demonstrates one technique for preserving binary compatibility when adding new data members. However, this technique requires planning ahead and reserving enough space for all future extensions. If there is not enough space reserved but enough space for a pointer, then the remaining space can be used to point to the rest of the data. For example:

```
class C : fix_size(12) { int x; int y; int i; int j; };
```

could become:

```
class C { int x; int y;
          struct {int i; int j;} * data; }
```

To do this, we modify the macro definition in Figure 4 to use the last bit of available space for the overflow pointer instead of returning an error. To the user of the class, the fact that some data members are stored in a separate object is completely transparent. In the above example, if `x` is an instance of class `C`, then data member `i` can be accessed using `x.i`. The full expansion of class `C` is something like:

```
class C { int x; int y;
          class Overflow {
              struct Data { int i; int j; };
              struct Data * ptr;
              Overflow() {ptr = malloc(sizeof(Data));}
              Overflow(const Overflow & o)
                  {ptr = malloc(sizeof(Data));}
              ~Overflow() {free(ptr);} };
          Overflow overflow;
          pseudo_member i int overflow.ptr->i;
          pseudo_member j int overflow.ptr->j; };
```

The key to making this work is the use of `pseudo_member` (which is built into the default class macro) to create pseudo members that behave like normal members for most purposes. This includes properly calling the constructor and destructor for the member if it has one. Thus, the members in `C::Overflow::Data` will get properly initialized even though `malloc`/`free` is used instead of `new` and `delete`.

In principle, the `fix_size` macro can work without the `pseudo_member` extension, but doing so will greatly increase the complexity of `fix_size` and implementing `pseudo_member` in the class macro was accomplished in around 6 lines of code. In addition, a closely related feature, `alias`, is useful for implementing other features such as anonymous unions. An `alias` is like a `pseudo_member` except that the constructor and destructor for the member are not called.

The enhanced `fix_size` macro can also be used to store all the private data, i.e. the "pimpl idiom," in a separate object by specifying a size of zero, which the `fix_size` macro would recognize as a special case.

Validation. Both previously mentioned techniques have been implemented in ZL as a macro library. All the end user needs to do is include the library, which will replace the class implementation with one that supports fixing the size. We have verified that the size does not change under various scenarios and hence binary comparability will be maintained.

4.2 Fixing the Size of the Virtual Table

Adding new virtual methods can break binary compatibility in essentially the same way as adding data members. Since the macro that implements classes uses another class to implement the vtable, all of the techniques previously discussed can easily be used to fix the size of the virtual table. To make this work, the ZL class macro provides a way to specify the implementation of the class used to implement the virtual table.

We have written a macro which uses the technique just described to allowing fixing the vtable size using the special syntax:

```
class X : fix_vtable_size(8) {...}
```

which will fix the vtable size to 8 bytes. We have also verified that the macro does indeed fix the size of the virtual table and hence maintains that aspect of binary compatibility.

4.3 A Better ABI

Adding new data members or methods breaks binary compatibility because the sizes of the class and vtable are needed at compile time.

The size of the class is needed when directly allocating an object on the stack, or when inlining one object into another. The first can be avoided by dynamically allocating the class on the heap. However, the second is a problem with most C++ ABIs as a typical C++ ABI defines class layout to be something like:

```
class Parent {...};
class Child { Parent parent; ...};
```

which inlines the parent in the child class. This means adding new data members to the parent class will break binary compatibility for any code that depends on the child class. We defined a new ABI to avoid this problem. Our new ABI defines class layout to be something like:

```
class Parent {void * child_ptr; ...};
class Child {
  Parent * parent_ptr; void * child_ptr; ...};
```

where the parent class is dynamically allocated when the child class is created, and `child_ptr` is used to downcast. This strategy preserves binary compatibility when new data members are added to the parent. A similar strategy is used for the vtable.

The code to implement the new ABI is under 60 lines of code. It overrides three methods from the core class macro; the method that adds the parent info to the user type was rewritten, and some additional information was added to every user type to include the child pointer.

We verified that the new ABI maintains binary comparability when adding new data members by creating a situation in which adding data members would cause problems with the more traditional ABIs. For example, in the following code:

```
class X {int x;}
class Z : public X {int z;}
```

adding a new data member, say `y`, to `X` will break binary compatibility with programs that use `Z` since the addition will change the offset of `z`. Therefore, accesses to the data member `z` will report an incorrect value. We verified that this was indeed a problem with ZL's default ABI, by setting the value of `z` with object code compiled against the new API (the one with the new `y` data member) but reading the value with object code compiled against the original API (without `y`) and verified that a different value was returned. We then did the same thing with the new ABI and verified that the same value was returned. We did a similar test to verify that adding new virtual methods will not break binary comparability.

For many purposes, this ABI can impose too much overhead. For example, each class must have a pointer to the child to support down casting, and virtual-method dispatch is slower. When binary compatibility is a primary concern, however, this ABI can be a good choice. Furthermore, since ZL can use more than one ABI at a time, a programmer can choose this ABI for just the parts of a program where the benefits in binary compatibility outweigh the costs in performance.

4.4 Matching an Existing ABI

Because classes are just user types to the compiler, it is possible to construct classes to match an existing ABI. This includes specialized ABIs which are really a C implementation of classes (such as done in GNOME [5]) or C wrappers around a C++ API (such as done in Aspell [6]). Doing so provides a more class-like interface to the C API. For example, ZL's macro API is a pure C API for simplicity; however, a more class-like interface is also provided. ZL provides a class-like interface to many of the API types including `Match`, `Syntax`, and `UnmarkedSyntax`. For example, instead of using `match_var(m, syntax x)`, one can use `m->var(syntax x)`. This is done by creating a user type `Match` that looks something like:

```
user_type Match { associate_type struct Match;
                  macro var(str, :this ths)
                      { match_var(ths, str);} };
```

4.5 Matching Other Compilers' ABIs

Just as it is possible to match a C ABI, it is possible to match other compilers' ABIs. It is even possible to use classes with different ABIs in the same program with some restrictions, which depend on fundamental incompatibilities between different ABIs. For example, while it is possible to mix classes with different ABIs through composition, doing so via inheritance is unlikely to work. This is due to fundamental differences in how inheritance is implemented, and in particular, how the vtable is laid out.

Fully matching a compiler's ABI requires careful reading of the specification, occasional reverse engineering, and careful testing to ensure compliance. As a proof-of-concept, we defined a new ABI by building on the existing class macro to pass the `this` parameter as a global variable. This implementation simulates passing the `this` parameter in a register, as the Microsoft C++ ABI does, as opposed to passing it as the first parameter, as GCC does. We then used both ABIs in the same program, and even embedded classes with one ABI in another via composition. The code to implement the new ABI was under 45 lines. The only methods from the core class macro that needed to be overridden were the ones involved with constructing and calling member functions—three in all.

5. Implementation Status and Performance

The current ZL prototype supports most of C and an important subset of C++. For C, the only major feature not supported is bitfields, mainly because the need has not arisen. C++ is a rather complicated language, and fully implementing it correctly is beyond the scope of our research. We aim to implement enough of C++ to demonstrate our approach; in particular, we support single inheritance, but currently do not support multiple inheritance, exceptions, or templates.

As ZL is at present only a prototype compiler, the overall compile time when compared to compiling with GCC 4.4 is 2 to 3 times slower. However, ZL is designed to have little to no impact on the resulting code. ZL's macro system imposes no run-time overhead.

The ZL compiler transforms higher level ZL into a low-level S-expression like language that can best be described as C with Scheme syntax. Syntactically, the output is very similar to fully expanded ZL as shown in Figure 1. The transformed code is than passed to a modified version of GCC 4.4. When pure C is passed in we are very careful to avoid any transformations which might affect performance. The class macro currently implements a C++ ABI that is comparable to a traditional ABI, and hence should have no impact on performance.

5.1 C Support

To demonstrate that ZL can support C programs, two well-known programs were compiled with ZL: bzip2 and gzip. Bzip2 was compiled without modifications, but gzip required some minor modification because it was an older C program and used some C syntax that is not a subset of C++: K&R-style function declarations were transformed into the newer ANSI C style, and one instance of `new` as a variable was renamed to `new_`.

Overall, compile times were 2 to 3 times slower with ZL in comparison to compiling with GCC 4.4. However, both programs compiled correctly, produced correct results, and had similar run times to the GCC-compiled versions.

5.2 C++ Support

To evaluate ZL's suitability to compile C++ programs, we chose to compile randprog [7], which is a small C++ program that gen-

erates random C programs. Randprog uses inheritance and other important C++ features, such as overloading and non-default constructors. In addition, it uses a few C++ features that ZL does not yet support, so we changed randprog in small ways to compensate. These changes include: reworking the command-line argument parsing, which used of a library that requires many modern C++ features; explicit instantiation of `vector` instances; changing uses of the `for_each` template function into normal `for` loops; and reworking some functions to avoid returning complex objects.

Randprog was verified to produce correct results by fixing the seed and comparing the generated program with a version of randprog compiled with GCC for several different seeds. It was also instrumented with Valgrind and found free of memory errors.

Overall compile time was around 2.5 times slower with ZL when compared to GCC 4.4. A direct run-time performance comparison is of limited usefulness, since ZL does not use the same C++ library as GCC, but the runtime performance of the ZL-compiled version of randprog was up to twice as fast as the GCC-compiled version.

6. Related Work

The problem of fragile and incompatible ABIs due to software and compiler changes is well known, and there have been several attempts to address the problem. To the authors' knowledge, ZL's approach of providing a small core language and letting everything else be defined as macros has not been tried before.

Binary Compatibility. The first serious attempt to solve the problem of fragile ABIs in C++ was in ΔC++ by Palay [2], but that ABI imposes a substantial performance penalty. Williams and Kindel developed a more sophisticated system with less overhead, known as the Object Binary Interface [3]. The Object Binary Interface is used only on request, and it allows for evolutionary steps, such as adding new public and protected methods and adding or removing private data members. However, it does not allow for changing the order or type of public data members; thus, it greatly reduces the problem of fragile ABIs, but does not entirely eliminate it. This ABI also imposes a higher cost when compared to the more traditional C++ ABI, and as such, is likely to affect performance, especially since all inheritance is implemented in a manner similar to how virtual inheritance is implemented in traditional C++ ABIs. Work on ΔC++ and the Object Binary Interface was done in the early 90s. Research on how to solve the problem in C++ since then is virtually nonexistent, most likely because of the inherent tradeoff between fragility and speed.

Some attempts have been made to standardize the C++ ABI between compilers for a given architecture. For example, the Itanium C++ ABI [1] aims to standardize the C++ ABI for the Itanium platform. This ABI is now used by GCC for all platforms towards the goal of providing a standard C++ ABI for GNU/Linux systems [8]. This effort has had some success, as the Intel C++ compiler also uses this ABI [9].

Since the problem of a fragile and incompatible ABIs was recognized as a serious issue that needed to be addressed, some newer languages, such as Java, specifically address the issue in the language specification. The Java concept of binary compatibility was first developed in SOM [10] and then later defined in the *Java Language Specification* (JLS) [11, 12]. In Java the ABI is completely specified in *The Java Virtual Machine Specification* [13], thus addressing the issue of incompatible ABIs.

Unfortunately supporting binary compatibility as specified in the JLS imposes a performance cost. Many Java compilers that support static compilation at first ignored binary compatibility in the interest of performance; one such compiler was the GNU Java Compiler, GCJ [14]. Later research by Yu, Shao, and Trifonov showed how to support static compilation and binary compatibility [12]. These techniques' were later integrated into GCJ [15].

Other Macro Systems. ZL's design philosophy is closely related to Scheme's [16] design philosophy of providing a small core language and letting everything else be defined as macros. The hygiene and module system are similar to Chez Scheme's `syntax-case` [4] and modules [17], respectively. There are numerous other macro systems for various languages, but apart from Scheme, few have the goal of allowing a large part of the language to be defined via macros. As such, they are either a macro system built on top of an existing language, or they lack procedural macros for general compile-time programming.

Maya [18] is a powerful macro system for Java. Maya macros (known as Mayans) can extend Java's LALR(1) grammar, in addition to being procedural and hygienic. OpenJava [19] and ELIDE [20] are similar to Maya but less advanced; neither of these systems support hygiene, and they do not support general syntax extensions. A procedural and hygienic macro system based on the Earley [21] parser is described in Kolbly's dissertation [22]. His system is similar to Maya in that macro expansion is part of the parsing process, yet more powerful as the Earley parser can handle arbitrary grammars rather than just the LALR(1) subset. Fortress [23] is a new language with hygienic macro support and the ability to extend the syntax of the language; however, Fortress's macros are not procedural. The Dylan [24] language has support for hygienic macros. In Dylan, macros are required to take one of three fixed forms: `def`, `stmt`, and `fun` call macros. The JSE system [25] is a version of Dylan macros adapted to Java. MS^2 [26] is an older, more powerful macro system for C. It is essentially a Lisp `defmacro` system for C. It offers powerful macros since they are procedural, but like Lisp's `defmacro` lacks hygiene.

Other macro systems include: ASTEC [27], which aims to be a safer C preprocessor; `<bigwig>` [28], which guarantees type safety and termination of the macro-expansion process; and MacroML [29], which has similar aims of `<bigwig>`.

Extensible Compilers. While macros are one approach to providing an extensible compiler, a more traditional approach is to provide an API that directly manipulates the compiler's internals, such as the AST. On the surface this approach may seem more powerful than a macro system, but a macro system can be equally powerful with the right hooks into the compiler.

Xoc [30] is an extensible compiler that supports grammar extensions by using GLR (Generalized Left-to-right Rightmost derivation) parsing techniques. Xoc's primary focus is on implementing new features via many little extensions, otherwise known as plugins. This approach has an advantage over most other extensible compilers in that the extensions to be loaded can be tailored for each source file. As such, Xoc provides functionality similar to that of traditional macro systems.

METABORG [31] is a method for embedding domain-specific languages in a host language. It does this by transforming the embedded language to the host language using the Stratego/XT [32] toolset. Stratego/XT supports grammar modifications using GLR parsing techniques. Polyglot [33] is a compiler front end framework for building Java language extensions. However, it uses a LALR parser, which means that perfectly valid grammar extensions can be rejected. JTS [34] is a framework for writing Java preprocessor with the focus on creating domain-specific languages. CIL [35] focuses on C program analysis and transformation, and as such, does not support grammar modifications. Again, as external tools, these systems all represent an approach different from ZL's support for extension within the language.

7. Conclusion and Future Work

Binary compatibility is a serious problem for software evolution in C++. C++ ABIs tend to be fragile because they are optimized for speed rather than robustness. Thus, library implementers have developed a number of programming idioms to help mitigate the problem. Due to the sheer number of idioms and the trade-offs involved, adding them as language extensions is infeasible. In addition, C++ ABIs differ between compilers, and hence, switching compilers often breaks binary compatibility.

We have solved this problem using ZL. ZL is a C++-compatible language in which high-level constructs, such as classes, are defined using macros over a C-like core language. ZL solves the problem of binary compatibility by using macros to automate the use of programming idioms that programmers would use to mitigate the problem. When programming idioms are not sufficient, ZL gives the programmer complete control over the ABI by providing a customizable class macro. The ZL macro system benefits library implementers and consumers who do not need to know the full details of how macros work, as library implementers can just use the macro libraries written by the tool implementers. At the same time, ZL makes the job of tool implementers easier when compared to a traditional compiler system.

ZL is currently a prototype compiler that can handle most of C and an important subset of C++. We have presented several working examples of how ZL can be used to mitigate the problem of binary compatibility and compiled a few real-world programs.

ZL is also a work in progress. Next steps include enhancing ZL to support more C++ features and matching the Itanium C++ ABI so that the ZL ABI will be compatible with modern versions of GCC. After that we hope to match other ABIs, such as the Visual C++ ABI, so that we can use a mixture of libraries, some compiled for GCC and some compiled for Visual C++, in the same program.

For the current implementation of ZL, see the ZL web page available at `http://www.cs.utah.edu/~kevina/zl/`.

Acknowledgments

We thank Eric Eide, Ryan Culpepper, and Jon Rafkind for their feedback on drafts of this paper.

References

[1] Itanium C++ ABI (revision: 1.86). `http://www.codesourcery.com/cxx-abi/abi.html`.

[2] Andrew Palay. C++ in a changing environment. In *Proc. USENIX C++ Technical Conf.*, 1992.

[3] Theodore C. Goldstein and Alan D. Sloane. The object binary interface: C++ objects for evolvable shared class libraries. In *Proc. USENIX C++ Technical Conf.*, 1994.

[4] R. Kent Dybvig, Robert Hieb, and Carl Bruggeman. Syntactic abstraction in Scheme. *Lisp and Symbolic Computation*, 5(4):295–326, 1992.

[5] GNOME. `http://www.gnome.org`.

[6] Aspell C API reference. `http://aspell.net/man-html/Through-the-C-API.html`.

[7] Eric Eide and John Regehr. Volatiles are miscompiled, and what to do about it. In *Proc. Intl. Conf. on Embedded Software (EMSOFT)*, 2008.

[8] A common C++ ABI for GNU/Linux. `http://gcc.gnu.org/gcc-3.2/c++-abi.html`.

[9] Intel C++ compiler man page. Available at `http://software.intel.com/en-us/intel-compilers/`.

[10] Ira R. Forman, Michael H. Conner, Scott H. Danforth, and Larry K. Raper. Release-to-release binary compatibility in SOM. In *Proc. OOPSLA*, 1995.

[11] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition*. Addison Wesley, 2005.

[12] D. Yu, Z. Shao, and V. Trifonov. Supporting binary compatibility with static compilation. In *Proc. Java Virtual Machine Research and Technology Symposium (JVM)*, 2002.

[13] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification, Second Edition*. Addison Wesley, 1999.

[14] GCJ web site. `http://gcc.gnu.org/java/`.

[15] Tom Tromey and Andrew Haley. GCJ: The new ABI and its implications. In *Proc. GCC Developers' Summit*, 2004.

[16] Michael Sperber (Ed.). The revised^6 report on the algorithmic language Scheme, 2007.

[17] Oscar Waddell and R. Kent Dybvig. Extending the scope of syntactic abstraction. In *Proc. POPL*, 1999.

[18] Jason Baker and Wilson C. Hsieh. Maya: multiple-dispatch syntax extension in Java. In *Proc. PLDI*, 2002.

[19] Michiaki Tatsubori, Shigeru Chiba, Kozo Itano, and Marc-Olivier Killijian. OpenJava: A class-based macro system for Java. In *Proc. 1st OOPSLA Workshop on Reflection and Software Engineering*, 2000.

[20] Avi Bryant, Andrew Catton, Kris De Volder, and Gail C. Murphy. Explicit programming. In *Proc. Conf. Aspect-Oriented Software Development (AOSD)*, 2002.

[21] Jay Earley. An efficient context-free parsing algorithm. *Commun. ACM*, 13(2):94–102, 1970.

[22] Donovan Kolbly. *Extensible Language Implementation*. PhD thesis, Univ. of Texas, Austin, 2002.

[23] Eric Allen, Ryan Culpepper, Janus Dam Nielsen, Jon Rafkind, and Sukyoung Ryu. Growing a syntax. In *Proc. Workshop on Foundations of Object-Oriented Languages (FOOL)*, 2009.

[24] Andrew Shalit, David Moon, and Orca Starbuck. *Dylan Reference Manual*. Addison-Wesley, 1996.

[25] Jonathan Bachrach and Keith Playford. The Java syntactic extender (JSE). In *Proc. OOPSLA*, 2001.

[26] Daniel Weise and Roger Crew. Programmable syntax macros. In *Proc. PLDI*, 1993.

[27] Bill McCloskey and Eric Brewer. ASTEC: a new approach to refactoring C. In *Proc. ESEC/FSE-13*, 2005.

[28] Claus Brabrand and Michael I. Schwartzbach. Growing languages with metamorphic syntax macros. In *Proc. Symposium on Partial Evaluation and Semantics-based Program Manipulation (PEPM)*, 2002.

[29] Steven E. Ganz, Amr Sabry, and Walid Taha. Macros as multi-stage computations: type-safe, generative, binding macros in macroml. In *Proc. Intl. Conf. Functional Programming (ICFP)*, 2001.

[30] Russ Cox, Tom Bergan, Austin T. Clements, Frans Kaashoek, and Eddie Kohler. Xoc, an extension-oriented compiler for systems programming. In *Proc. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008.

[31] Martin Bravenboer and Eelco Visser. Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions. In *Proc. OOPSLA*, 2004.

[32] Eelco Visser. Program transformation with Stratego/XT. Rules, strategies, tools, and systems in Stratego/XT 0.9. In Lengauer et al., editor, *Domain-Specific Program Generation*, Lecture Notes in Computer Science, June 2004.

[33] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for java. In *Proc. Conf. Compiler Construction*, 2003.

[34] D. Batory, B. Lofaso, and Y. Smaragdakis. Jts: Tools for implementing domain-specific languages. In *Proc. Intl. Conf. Software Reuse (ICSR)*, 1998.

[35] George C. Necula, Scott McPeak, S. P. Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Proc. Conf. Compiler Construction*, 2002.