Toward Modular, Multidimensional Extensibility in Programming Languages

Project Description

1 Introduction

Languages that support the development of reusable and extensible components provide important benefits. Such languages foster a programming style based on combining and extending existing abstractions rather than building programs from scratch. As a simple example, programs typically rely on third-party libraries, which may be used in contexts and extended in ways that were not anticipated by the libraries' implementers. Similarly, independent teams of developers can create their respective components in isolation, later linking them into the final application. Such languages also provide better support for software evolution and maintenance, allowing existing components to be updated and extended non-invasively. Finally, in the context of the Internet, support for reusability and extensibility become even more important. For example, programs may employ a "plug and play" approach, dynamically linking components that were not known at static compile time.

Existing language paradigms typically support modular reuse and extensibility along one dimension, but fail to support extensibility along other dimensions [Cook 90, Odersky & Wadler 97, Appel 97, Findler & Flatt 98, Krishnamurthi et al. 98, Wadler 98, Zenger & Odersky 01]. For example, classes in a standard object-oriented (OO) language are a reusable abstraction, easily shared across programs. Further, OO languages support *differential programming*, allowing new classes to be defined as extensions to existing classes. An existing class need not anticipate the extensions required by its subclasses, and the existing class need not be modified. Thus, subclassing is one dimension of extensibility well supported by OO languages. On the other hand, existing classes cannot be easily extended with client-specific methods, unless such extension is anticipated by the original class implementer. Without such anticipation, the original class must be modified in place, thereby requiring access to the source code, polluting the interface of the class as seen by all clients, and necessitating retypechecking and recompilation of the class. Therefore, this dimension of extensibility, the addition of new operations to an existing type, is not supported by OO languages. A third dimension of extensibility is the ability to modify or augment existing operations from the outside, without invasively editing the code in place. OO languages can be viewed as supporting a very stylized form of this kind of extensibility, allowing an existing operation to be augmented with specialized behavior for a particular new subclass of the operation's receiver type. More expressive forms of operation update require existing methods to be updated in place and recompiled.

In the functional programming language paradigm, modules are a common reusable entity. In ML, modules may contain datatype declarations to represent an abstraction, along with some associated functions. Clients can easily create new functionality for the abstraction, without modifying the original module. Thus, functional languages support extension via adding new operations to an existing data type, unlike OO languages. However, if a client-specific data variant is required, the original module must be modified in place, augmenting the original datatype declaration and extending each of the datatype's functions to handle the new variant. Consequently, functional languages fail to support the dimension of extension via subclassing, in contrast to OO languages. Functional languages also fail to support extension via update of existing operations, instead requiring existing functions to be modified in place.

The examples of the OO and functional paradigms illustrate a common problem of current programming languages. Each language supports a particular dimension of reusability and extensibility well, while providing little or no support for other dimensions of extensibility. If a client requires a form of extensibility that is not supported by the language, existing components must be updated in place. Aside from being error-prone and expensive, requiring retypechecking and recompilation of the component, it is unreasonable in many cases to assume access to a component's source code. In such cases, the client must choose to forgo the necessary extensions or write the needed abstraction from scratch.

Several proposals have been made to address these limitations of existing languages. For example, *multimethods*, found in languages such as Common Lisp [Steele Jr. 90], Dylan [Dyl92, Shalit 96], and Cecil [Chambers 92, Chambers 93a], naturally generalize the methods of traditional object-oriented languages, allowing the appropriate method to invoke for a message send to depend on the run-time classes of any subset of the message arguments. This contrasts with the situation in traditional object-oriented languages, where method selection can depend only on the run-time class of the single receiver argument. For example, both arguments of the following equality method are *specialized* via the `@class` syntax, signifying that the run-time class of both arguments to the method must be `Point` or a subclass:

```
boolean equal(p1@Point, p2@Point) { return p1.x = p2.x and p1.y = p2.y; }
```

Any subset of a method's arguments can be specialized in this way, and the specialization pattern of one method is independent of that of any other method. We have also explored a generalization of multimethods [Ernst et al. 98b] that extends multimethod dispatching to include predicated dispatching [Hamer et al. 90, Chambers 93b] and function definition by cases as found in languages with pattern-matching like ML [Milner et al. 90, Milner et al. 97] and Haskell [Hudak et al. 92], allowing dispatching to be based on properties of the arguments other than their run-time class.

Because multimethods eliminate the distinction between the receiver and the other arguments of a message send, multimethods are naturally defined outside of classes. Instead, modules are used to structure both classes and multimethods. For example, if a `ColoredPoint` subclass of `Point` is defined later, we can augment the equality functionality for `Points` without modifying in place any existing classes or methods, by creating the following multimethod in a new module:

```
boolean equal(p1@Point, p2@ColoredPoint) {
    return p1.x = p2.x and p1.y = p2.y and p2.color = "black"; }
```

Adding this functionality in a traditional OO language would require modification of the `equal` method inside the `Point` class. Structuring methods outside of their classes also makes it easy to add new operations non-invasively. For example, if a particular client of points requires the ability to print them, the client can simply create a new module containing a `print` method for `Points`. We refer to the ability to add a new operation modularly to an existing class as an *open class* idiom.

When viewed from the perspective of extensibility, it becomes clear that multimethods enable an OO language to support multiple dimensions of modular extensibility: 1) new subclasses can be added modularly, as usual; 2) new multimethods can be added to existing classes modularly; and 3) new multimethods can be added modularly to augment existing operations, in the case that the operation's functionality can be divided up based on the properties of its arguments.

As another example of an extensibility mechanism, aspect-oriented programming [Kiczales et al. 97] provides support via *aspects* for concerns that "crosscut" the natural decomposition mechanisms of a programming language. AspectJ [Kiczales et al. 01] is an aspect-oriented extension to Java [Gosling et al. 96, Arnold & Gosling 96]. As with multimethods, AspectJ's aspects support open classes. In addition, aspects can specify pieces of code to be run at certain *join points* in the execution of the program. For example, the following aspect defines a *pointcut* representing all calls to `equal` on `Point` or a subclass and extends such calls with tracing functionality, without modifying existing code:

```
aspect TraceEqual {
    pointcut traceEqual:
        calls(boolean Point.equal(Point));
    before(): traceEqual() {
        System.out.println("Entering equal");
    }
    after(): traceEqual() {
        System.out.println("Exiting equal");
    }
}
```

In this way, aspects can extend existing classes and methods with new code non-invasively, in flexible ways that are not possible to express using multimethods.

Unfortunately, while multimethod-based languages and AspectJ support multiple dimensions of extensibility and allow code to be modularized in fine-grained ways, these languages lose a critical benefit of traditional languages: the ability to safely reason about a component in isolation. For example, a class in a standard OO language can be typechecked and compiled given only the interfaces of the classes it refers to; the source code for those classes need not be available, and nothing need be known about the rest of the program. Classes can then be combined together at link time, without fear that a type error within one of the components will be discovered only then. In contrast, a module in a multimethod-based language and an aspect in an aspect-oriented language can only be safely typechecked and compiled when the *entire* program is available. For example, aspects are not typechecked and compiled separately. Instead, aspects are globally translated away, with appropriate code snippets moved from the aspects to the relevant classes and methods, and subsequently ordinary Java typechecking and compilation is performed. Any typechecking performed on individual aspects is only provisional, since it is always possible for independent but incompatible aspects to be linked into the same application, creating a program that does not typecheck.

This loss of modular reasoning enormously limits the benefits of the extra expressiveness provided. Consider the motivating scenarios for reusable and extensible components, described earlier. Without modular reasoning, a library must be compiled once per context in which the library is used, which is expensive and requires the library's source code, a requirement that the extra expressiveness was supposed to alleviate. Independent teams of developers can only be sure that their respective components are type-safe at link time, when the entire application is available.

In previous work, we defined a core language called *Dubious* that explored a range of modular typechecking algorithms for multimethod-based languages, representing various points in the expressiveness/modularity tradeoff [Millstein & Chambers 99, Millstein & Chambers]. One of those type systems, called System M, supported completely modular typechecking while still providing many useful multimethod programming idioms. Recently, we employed our experience with Dubious and System M in the design of *MultiJava*, an extension to Java supporting multimethods and open classes [Clifton et al. 00]. In addition to preserving Java's modular typechecking scheme, we also defined an implementation strategy that compiles MultiJava's compilation units to regular Java bytecode, file by file, using a generalization of Java's existing modular compilation strategy.

We propose to build on this work to develop language support for fine-grained extensibility along multiple dimensions, while at the same time preserving safe, modular reasoning about components. First, we propose to design and implement a generalization of MultiJava, called Relaxed MultiJava, that gives the programmer greater freedom to manage the tradeoff between modularity and expressiveness. Because MultiJava currently has purely modular typechecking and compilation, it places some conservative restrictions on the kinds of extensions that one compilation unit can make to an existing compilation unit. While MultiJava is strictly more expressive than ordinary Java, there are useful programming idioms that MultiJava's type system does not allow. Relaxed MultiJava will relax the restrictions imposed by the conservative static type system in the cases where MultiJava detects only the *potential* for a link-time error, not an *actual* error; in these cases, the compiler will issue warnings rather than errors. A special Relaxed MultiJava class loader will verify incrementally, as each class is loaded, whether or not any of the potential errors actually occur in the linked program. In this way, Relaxed MultiJava will provide the same modular guarantees as MultiJava while allowing a much wider range of idioms to be expressed safely. This extension requires developing new techniques for fast typechecking of multimethods and open classes that is integrated with Java's lazy class loading scheme.

Complementing the work on Relaxed MultiJava, we propose to develop new compilation techniques for MultiJava-like languages. MultiJava's current implementation strategy supports modular file-by-file compilation, but for methods added via the open class idiom, the code generated to perform method dispatch is not as efficient as regular Java method calls. In addition, the MultiJava compilation strategy is not sufficient to handle the more expressive idioms allowed by Relaxed MultiJava. We will investigate techniques for dynamically generating, at class load time, more efficient dispatchers based on global information about the current set of loaded classes and methods. We expect to extend techniques we developed previously [Chambers & Chen 99] for the generalized multimethod model mentioned above [Ernst et al. 98b] to support incremental update and dynamic extension. We may also explore extensions to the Java virtual machine that can allow multimethod-based languages to execute more efficiently.

We also propose to design and implement an extension to ML called *Extensible ML* (*EML*) that smoothly supports multiple dimensions of extensibility within the context of an advanced functional programming language. First, we will generalize ML's datatypes to be hierarchical and extensible, thereby able to simulate class hierarchies. Second, we will generalize ML's functions to be extensible, allowing new cases to be added to a function from outside modules. In this way, function cases faithfully simulate the abilities of OO-style methods, including method overriding. As with MultiJava, the EML type system will adapt key ideas from our foundational work with Dubious, but there are several new technical challenges to address. First, OO-style dispatching must be seamlessly integrated with ML-style pattern matching. This problem was addressed to some extent by our previous work on generalized dispatching [Ernst et al. 98b], but that work required global typechecking for safety. Second, we must also integrate our modular typechecking schemes with ML-style parametric polymorphism, possibly also providing support in EML for bounded and F-bounded polymorphism [Cardelli & Wegner 85, Canning et al. 89] over the data variant hierarchies. Third, we will study how to provide some form of ML-style type inference in the language. Finally, classes interact in powerful ways with ML's module system. ML's functors can be used to parameterize a class definition by its superclass and associated operations, as is used in mixin-style programming [Moon 86, Bracha & Cook 90, Steyaert et al. 93, Flatt et al. 98, Smaragdakis & Batory 98] and role-based programming [Reenskaug &

Anderson 92, Harrison & Ossher 93, VanHilst & Notkin 96]. It is challenging to retain modular typechecking in the presence of these expressive idioms.

Once these developments are well understood, we propose to develop modular typechecking and compilation strategies for still-more-flexible approaches for multidimensional extensibility, including AspectJ's pointcut and method combination abilities [Kiczales et al. 01] and the "hyperslice" merging which is fundamental to Hyper/J [Ossher & Tarr 00], a technology based on subject-oriented programming [Harrison & Ossher 93]. To this end, we plan to explore richer interfaces for program fragments, that provide precise information about the expectations and assurances of the code fragments being manipulated. These interfaces can then be used to safely reason about fragment composition in a modular setting.

To help assess our work, we are planning to undertake several case studies on real programming systems. We will translate several large Java programs into Relaxed MultiJava, to see how well our constructs can improve the structure and future extensibility of these programs. We have identified several large, locally developed Java programs that appear to be excellent candidates for study. We will also find several interesting ML programs and translate them to EML. In addition, we plan to translate key parts of our existing large body of Cecil code (including a 125,000-line optimizing compiler that makes heavy use of multimethods, open classes, and parametric polymorphism) to EML, to evaluate how well EML's modular typechecking works for complex, real programs.

We will release our Relaxed MultiJava and EML implementations to the outside research community.

2 Previous Work

In this section, we summarize the state of the art in language support for modular, multidimensional extensibility.

2.1 Object-Oriented Languages

As mentioned in the introduction, multimethods provide certain forms of modular extensibility that are not supported by traditional OO languages. Common Lisp [Steele Jr. 90] and Dylan [Dyl92, Shalit 96] provide very flexible multimethods, including support for method combination and the dynamic extension of existing operations with new multimethods. However, these languages are not statically typed. Polyglot [Agrawal et al. 91], Cecil [Chambers 92, Chambers 93a], Kea [Hamer et al. 90, Hosking et al. 90, Mugridge et al. 91], BeCecil [Chambers & Leavens 97], and Tuple [Leavens & Millstein 98] support multimethods and open classes with static typechecking, but the whole program must be available to perform this typechecking. An efficient global typechecking algorithm has been developed for languages with multimethods [Chambers & Leavens 94, Chambers & Leavens 95a] and later extended to support a flexible polymorphic type system [Litvinov & Chambers 98, Litvinov 98] and a general dispatching model unifying multimethods, predicate classes, and ML-style pattern-matching [Ernst et al. 98b]. Encapsulated multimethods [Castagna 95, Bruce et al. 95] and parasitic multimethods [Boyland & Castagna 97] support modular static typechecking but disallow the open class idiom and modify the natural symmetric semantics of multimethods to treat the receiver as more important than the other arguments. The λ&-calculus [Castagna 97] supports modular static typechecking and symmetric treatment of multimethods but requires all multimethods implementing a given operation to be defined as a unit, similar to function cases in ordinary (non-extensible) functions.

AspectJ [Kiczales et al. 01] is an extension of Java based on aspect-oriented programming [Kiczales et al. 97]. AspectJ provides *aspects* as an alternative to traditional subclass-based extensibility. An aspect may introduce new methods into existing classes without needing to modify those classes in place, thus supporting open classes. In addition, aspects can include methods that are combined in flexible ways with the existing methods of a class, without modifying existing code. The logging example in the introduction, for example, uses the "before" and "after" method combination mechanisms. An expressive language for specifying sets of *join points* provides programmers explicit control over where an aspect's extensions will be used. Unfortunately, aspects are not typechecked or compiled modularly. Instead, the whole program is preprocessed as a unit to yield a version of the program where the aspects have been woven into the appropriate classes. At this point, regular Java typechecking and compilation is used, and type errors may arise at this point that cannot be detected by looking at any one aspect in isolation. Therefore, source code is required for all classes extended by aspects, and retypechecking and recompilation of the modified versions of these classes are required if aspects are changed.

Hyper/J [Ossher & Tarr 00] is an extension of Java that provides "multidimensional separation of concerns," based on subject-oriented programming [Harrison & Ossher 93]. Hyper/J allows almost arbitrary fragments of classes, called *hyperslices*, to be created and combined in flexible ways to form complete classes. As a simple example, the open class idiom can be performed by simply creating a hyperslice containing the new methods. Hyperslice combination is directed by a set of composition rules. For example, two hyperslices containing methods of the same name could be combined by having one of the methods shadow the other or by concatenating the method bodies into a single method using some form of method combination. Hyper/J also supports *on-demand remodularization,* whereby existing classes may be decomposed into hyperslices (including overlapping hyperslices) after the fact; these hyperslices can then be re-composed in new ways to form variants on the original class. Similarly to aspects, hyperslices are not modularly typechecked and compiled. Instead, typechecking and compilation is performed after hyperslice combination, at which point the program consists of a set of ordinary Java classes.

2.2 Functional Languages

OML [Reppy & Riecke 96] is an object-oriented extension to ML. OML's `objtype` construct is a form of hierarchical, extensible datatype, allowing new variants to be defined without modifying existing code. However, OML retains ordinary ML-style functions, which are not extensible. Therefore, existing functions cannot be updated as new data variants are defined. Instead, OML allows methods to be defined inside of objtypes. Like ordinary OO methods, they can be overridden in new sub-variants, but they cannot be added to an existing objtype without modifying the original declaration. Hence, programmers must choose between writing code using traditional ML datatypes and functions versus using objtypes and methods; the kinds of future extensibility allowed greatly depend on this choice. In a similar vein, Objective ML [Rémy & Vouillon 97] extends ML with a class construct. As with OML, programmers must choose between the kind of extensibility offered by traditional ML constructs and the incomparable extensibility offered by the new class constructs [Leroy 99].

Zenger and Odersky [Zenger & Odersky 01] describe an alternative form of extensible datatype. Extending a datatype has the effect of creating a new datatype that subtypes from the original one, rather than updating the original datatype definition. To ensure that existing functions will handle the new variants, all functions on extensible datatypes must include a *default* case. If the programmer wishes to update an existing function to handle a new variant, a new function must be created that inherits the existing function cases. Because adding new variants and function cases is functional, resulting in the creation of a new datatype or function, existing clients of the datatype or function must be modified to use the new version, and then re-compiled. As a result, this kind of extensible datatype does not faithfully simulate the effect of declaring a new subclass with some overriding method definitions.

Similar to OML, $ML_{\leq}$ [Bourdoncle & Merz 97] generalizes ML datatypes to be hierarchical and extensible. Methods are simulated via function cases, whose patterns encode OO-style dynamic dispatching. $ML_{\leq}$ supports a sophisticated type system that includes polymorphic subtype constraints on methods. $ML_{\leq}$ patterns do not support full ML-style pattern matching. Further, the authors do not consider the interaction of extensible datatypes with ML's module system. Most importantly, $ML_{\leq}$ assumes that all function cases of a function are defined as a unit. A relaxation is sketched supporting extensible functions, but this relaxation requires global typechecking.

3 Proposed Work

Our research plan includes four main components: the relaxation of MultiJava's type system to handle more expressive programming idioms and the development of efficient dispatching techniques for MultiJava-like languages, the development of the Extensible ML programming language, the assessment of these languages via several case studies, and the exploration of modular typechecking and compilation for other powerful language extensibility mechanisms such as AspectJ and Hyper/J. These components are described in the following subsections.

3.1 Relaxed MultiJava

3.1.1 MultiJava Strengths and Weaknesses

MultiJava allows several programming idioms that are difficult or impossible to express in ordinary Java. For example, consider a `Point` class that has a `ColoredPoint` subclass. Suppose that a client requires a specialized way of printing points. In MultiJava, the client can add the new operation by creating a new compilation unit with the following *top-level method declarations*:

```
String Point.printString() { return "(" + getX() + "," + getY() + ")"; }
String ColoredPoint.printString() {
   return "(" + getX() + "," + getY() + "," + getColor().toString() + ")"; }
```

This new compilation unit can be typechecked and compiled completely modularly, without access to the source code for the point hierarchy. The client can then invoke the new operation just as any other operation on points is invoked. For example, if `p` is an instance of `Point` or a subclass, then `p.printString()` will return the appropriate string representation of `p`.

This solution compares favorably to the workarounds used to modularly add new operations to existing classes in traditional OO languages. One possibility is to create new subclasses, like `PrintablePoint` and `PrintableColoredPoint`, containing the new operation. Aside from being more tedious and error-prone than MultiJava's solution, this solution does not allow the client to make use of existing `Point` and `ColorPoint` instances, as they will not support the new operation. Therefore, existing code must be modified to create instances of the new subclasses instead and to adjust the static types along the dataflow paths of the new subclasses, and then the new code must be retypechecked and recompiled. Alternatively, the visitor design pattern [Gamma et al. 95] allows new operations to be added to existing classes without modifying existing code. However, the pattern is tedious and error-prone, relying on a complex protocol between the original class hierarchy and a separate hierarchy of "visitors." Further, the implementer of the original class hierarchy must plan ahead for this extensibility by providing the appropriate "hooks" for visitors, or else the new operations cannot be added modularly. Finally, the visitor pattern gains the ability to modularly add new operations at the cost of losing the ordinary OO ability to add new subclasses without modifying existing code.

Despite these advantages of MultiJava's open classes and top-level methods over previous approaches, there are useful idioms that are not expressible in MultiJava. For example, suppose that the `Point` class were declared `abstract`, so that `Point` instances cannot be created and the class need not be fully implemented. In this case, it would be natural to make the first `printString` method above abstract as well, relying on each concrete subclass of `Point` to define an appropriate method. This would be fine if `printString` were implemented inside of `Point`. However, if `printString` were written in a separate compilation unit, then it would not be safe to allow the method to be abstract. For example, suppose that a subclass `Point3D` of `Point` is defined at some point, independently and separately from the definition of `printString`. The implementer of the compilation unit containing `printString` was not aware of `Point3D`, so no appropriate method was written. Likewise, the `Point3D` implementer is unaware of the `printString` operation. Therefore, at compile time, neither compilation unit observes the other, so both typecheck successfully (assuming we allow `printString` on `Points` to be left abstract). However, if a `Point3D` instance is ever sent the `printString` message, a run-time error will occur, because no appropriate method will be found. It is precisely to prevent problems of this kind that top-level methods cannot be abstract in MultiJava. In the `printString` example, forcing the `Point` class to have an appropriate "default" implementation ensures that there will always be an appropriate implementation for any unseen subclasses of `Point`, such as `Point3D`.

For operations that have a natural default behavior, MultiJava's restriction is not terribly onerous. However, in many cases there is no reasonable default semantics for an operation. In the `printString` example, the printable representation of `Point3D` instances will not reveal the 3D point's z-coordinate value, which may be important for some clients. For other operations, there may be no alternative but for the default implementation to simply throw an exception, because the associated abstract class simply does not have a rich enough interface (consider writing `Point`'s `printString` method if there were no `getX` and `getY` operations defined on `Points`, for example). In this case, the modular type safety provided by MultiJava is unsatisfying, because a run-time error will occur if the default method is ever invoked.

3.1.2 Relaxed MultiJava

To address these limitations, we propose to relax MultiJava's type system, yielding the *Relaxed MultiJava* system. For example, to remove the limitation described above, `Point`'s `printString` method will be allowed to be declared `abstract`. The existence of a top-level abstract method will generate a compile-time *warning*, rather than a fatal *error*. For safety, we must then check at class-load time whether the warning is indicative of a "real" error. In our example, an error will be signaled if `Point3D` is loaded, because no appropriate `printString` method is found. On the other hand, if the only concrete subclasses of `Point` ever loaded either inherit from `ColoredPoint` or include their own `printString` method, the program will remain typesafe and no error will be signaled. For clients that combine independently written compilation units such as `printString` and `Point3D`, Relaxed MultiJava will allow the client to provide additional methods to fill in any gaps, in this case writing a new compilation unit that defines a `printString` method for `Point3D` objects.

Other restrictions imposed by MultiJava's type system, such as constraints on how multimethods can be written to avoid ambiguous definitions, will be relaxed in a similar fashion: errors will be replaced with warnings plus load-time checks. These relaxations give programmers fine-grained control over the tradeoff between expressiveness and modularity. If no warnings are signaled at compile time, then the programmer has the same guarantee of modular type safety as in regular MultiJava. If a warning is signaled at compile time, the programmer has two options. He can choose to resolve the warning (in our first example, by adding a default `printString` implementation) immediately, as he would be forced to do in regular MultiJava. Or he can choose to leave the code as is, under the assumption that the warning will never be manifested as a real error at load time. If it turns out to be a real error, this will be detected when the offending compilation unit is loaded. Clients that combine independent libraries are also able to resolve errors by providing additional code, e.g. to fill in gaps, which would not be possible within the libraries themselves.

To implement Relaxed MultiJava, we will create a specialized class loader that will perform the necessary checks upon loading each class. To communicate to the loader what checks are necessary, we will augment the Java class file format with MultiJava-specific information. Whenever a warning is signaled at compile time, the necessary information will be added to the class file. In our first example, the class file containing the `printString` methods would convey the fact that the operation is unimplemented for `Point`. Consequently, after the class loader has loaded this file, it will know to check concrete subclasses of `Point` for an appropriate `printString` method.

Because class loading occurs dynamically in Java, it is particularly important for these load-time checks to be fast. There are several reasons to expect this to be the case. First, regular MultiJava should be expressive enough in the common case, so we would not expect too many warnings to be signaled in a program. Second, even when load-time checks are required, most of the checking has already occurred at compile time. In our example, there is no need to re-check `ColoredPoint` and subclasses, since the regular modular checks verified the existence of an appropriate `printString` method in this case. Finally, we can leverage Java's lazy class loading scheme to amortize the cost of the checks across multiple class loads. Accordingly, we plan to develop incremental algorithms for performing MultiJava's checks, that can be seamlessly integrated into the standard class-loading process.

3.1.3 Efficient Multimethod Dispatching

The previous MultiJava work included a strategy for compiling top-level methods into regular Java bytecode, file by file. Unlike the compilation strategies for AspectJ and Hyper/J, the classes that the top-level methods augment do not need to be recompiled, and in fact top-level methods can augment regular Java classes, including built-in system classes which cannot be updated by AspectJ or Hyper/J. However, the method dispatching speed for top-level methods is slower than for regular Java method calls. In addition, the MultiJava compilation strategy is not sufficient to handle the more expressive idioms allowed by Relaxed MultiJava.

We propose to develop new compilation techniques for MultiJava-like languages. The key idea is to dynamically generate a specialized dispatch routine at class load time. The class loader has global knowledge of the set of loaded classes and top-level methods, and so it can generate more efficient dispatchers than could a static compiler with only a partial view of the program. We plan to build on techniques for generating efficient dispatchers we developed previously [Chambers & Chen 99]. We will extend them to support incremental recomputation of dispatchers (to minimize the cost of regenerating dispatchers as a new class or top-level method is loaded) and incremental update of previously generated dispatcher data structures and code. These extensions will be useful beyond the context of MultiJava.

Finally, we may explore extensions to the Java virtual machine that can allow multimethod-based languages to execute more efficiently, shifting the dispatcher generation work into the virtual machine itself where more flexible strategies are possible.

3.2 Extensible ML

3.2.1 Extensible Datatypes and Functions

As mentioned in the introduction, functional languages suffer from the converse extensibility problem of OO languages: new operations are easily added to existing datatypes without modifying existing code, but new data variants cannot be added without modifying the original datatype declaration along with all functions defined on the datatype. We propose to develop the *Extensible ML* (*EML*) language to address this issue. The basic idea is to generalize ML datatypes to be *hierarchical* and *extensible*. A hierarchical datatype would allow variants to be organized into an inheritance tree, with the function cases pattern-matching on one variant inheritable by extending sub-variants. An extensible datatype would allow new variants to be added to an existing hierarchy from the outside, without having to modify the definition of original hierarchy in place. To enable extensible datatypes to faithfully simulate OO class hierarchies with method overriding, ML functions over an extensible datatype must also be extensible, so that as new data variants are declared, existing functions can be updated with the corresponding new function cases. Hierarchical, extensible datatypes and extensible functions will be the core constructs of EML. Making ML functions extensible is the key contribution over previous attempts to add OO-style extensibility to ML discussed in section 2.2.

The following EML module implements a simple hierarchy of points:

```
structure PtMod = struct
   abstract class Pt of {};
   class CartPt extends Pt of {x:real,y:real}
   class Origin extends Pt of {};
   fun getX: Pt -> real
   extend fun getX (CartPt {x=x,y=_}) = x
   extend fun getX (Origin _) = 0.0
   fun getY: Pt -> real
   extend fun getY (CartPt {x=_,y=y}) = y
   extend fun getY (Origin _) = 0.0
end
```

A `class` is EML's notation for an extensible datatype. Each class declares its local representation type (a record of instance variables) after the `of` keyword, similarly to how the representation type associated with each data variant in ML is specified. The class's local representation type is appended to the representation of its superclasses (if any) to form the full representation of instances of the class. The abstract class `Pt` is the top of the point hierarchy, with two concrete subclasses representing Cartesian points and the origin, respectively. A regular ML datatype declaration becomes syntactic sugar for a set of class declarations. For example, we can equivalently define the point hierarchy above via a regular ML datatype declaration as follows:

```
datatype Pt = CartPt of {x:real,y:real} | Origin of {}
```

To make functions extensible, we break an ordinary ML `fun` declaration into two pieces. EML's `fun` declaration simply declares a function, while `extend fun` adds cases to a previously-declared function. For example, the `PointMod` module declares an `getX` function of type `Pt -> real`. The `getX` function has two associated function cases. The `extend fun` declaration *imperatively* updates an existing function, rather than creating a brand new function. This semantics allows extensible functions to faithfully model OO-style methods of the same name, which conceptually update a common "generic function." As a syntactic sugar, we can equivalently define the `getX` function and associated cases via a regular ML function declaration as follows:

```
fun getX (CartPt {x=x,y=_}) = x
  | getX (Origin _) = 0.0
```

A function case's pattern combines ML-style pattern matching with OO-style dynamic dispatching. For example, the first `getX` function case in the example is applicable to arguments that are dynamically instances of `CPt` or a subclass, whose representation additionally matches the given record pattern. As in OO languages, a "best-match" semantics is used to determine which function case a given function application invokes; the order of function cases is irrelevant. This contrasts with ML's "first-match" semantics for function cases, which does not generalize naturally to handle extensible datatypes and functions. In such a setting, typically the more-specific function cases are written after the less-specific ones, as new data variants are defined, which is the opposite of what is required for a "first-match" semantics.

As in regular ML, it is easy to add new functions to existing datatypes without modifying existing code. For example, we can add a function for computing the Cartesian distance between two points in a new module:

```
structure DistMod = struct
   open PtMod
   fun dist: (Pt * Pt) -> real
   extend fun dist (p1, p2) =
      sqrt(pow(getX(p1)-getX(p2),2.0) + pow(getY(p1)-getY(p2),2.0))
   extend fun dist (Origin _, Origin _) = 0.0
end
```

The ML style naturally supports multimethod dispatching, since there is no notion of a receiver argument. For example, the second `dist` function case above is only applicable dynamically if both arguments are an instance of `Origin` or a subclass.

Unlike regular ML, in EML it is equally straightforward to add new data variants to existing datatypes without modifying existing code. For example, we can create a new module to define a variant representing polar points:

```
structure PolarMod = struct
   open PtMod
   class PolarPt extends Pt of {rho:real, theta:real}
   extend fun getX (PolarPt {rho=r, theta=t}) = r * cos(t)
   extend fun getY (PolarPt {rho=r, theta=t}) = r * sin(t)
end
```

The `PolarPt` variant is defined as a subclass of the existing `Pt` class. In addition, the existing `getX` and `getY` functions are augmented with new function cases to handle polar points.

3.2.2 Modular Typechecking

A critical requirement of EML, as in MultiJava, is the ability to perform typechecking completely modularly. Specifically, we must be able to typecheck each EML module *M* given only the interfaces (*signatures*) of the modules to which *M* refers. For example, `DistMod` should be able to be safely typechecked and compiled given `PtMod` but not `PolarMod`. In this way, the `dist` function must be assured to be completely and unambiguously implemented, even though the relevant `PolarPt` variant is not available. As with MultiJava, we can base our modular typechecking algorithm for EML on Dubious's System M. For example, the first `dist` function case in `DistMod` acts as the ML version of a "default method," applicable to any two instances of `Pt` or a subclass. The existence of this function case therefore ensures that the unseen `PolarPt` variant is completely and correctly handled.

However, several new technical issues must be addressed in the context of modular typechecking for EML. First, EML's function cases generalize both ML-style pattern matching and multimethod dispatching. This generalized dispatching has been previously explored [Ernst et al. 98b], but two critical issues were not addressed. First, because patterns can be nested arbitrarily deep, care must be taken to perform typechecking that is safe but not overly conservative. Previous work does not take into account the level of nesting in the patterns being checked, instead providing a general algorithm that is conservative but will often signal a type error where none exists. Second, previous work provided only a global typechecking algorithm, relying fundamentally on access to all relevant function cases and data variants. We will adapt Dubious's restrictions for modular type safety to handle the generalized patterns.

EML's modular typechecking algorithm must also be able to handle ML-style parametric polymorphism; the type systems of both Dubious and MultiJava include only subtype polymorphism, not parametric polymorphism. Because EML's datatypes are hierarchical, it would also be useful to support (F-)bounded polymorphism. It appears from our initial exploration that, given some well-formedness restrictions on the structure of the inheritance graph over classes (EML will allow both single inheritance of implementation and multiple inheritance of interface, as in Java), the presence of parametric polymorphism is largely orthogonal to the extensibility of datatypes and functions.

ML supports type inference for functions and expressions, but it does not support subtyping. Extending ML-style polymorphic type inference to support subtyping is the subject of ongoing research [Fuh & Mishra 90, Hoang & Mitchell 95, Nordlander 99, Pierce & Turner 00]. We will investigate techniques to provide at least some limited form of type inference for EML. Type inference is only a convenience, so partial type inference is a practical compromise that alleviates some of the programming task while allowing multidimensional extensibility.

We will formally specify the static and dynamic semantics of EML, and we will prove the type system sound.

3.2.3 Modules

EML will include an ML-style module system [MacQueen 84, Milner et al. 97]. A key challenge is retaining modular typechecking for *functors*, which are a form of parameterized module. In the presence of extensible datatypes and functions, functors provide the ability to program mixins, classes parametrized by their superclass, and related idioms. For example, consider the following code:

```
signature MyPtSig = sig
    class MyPt extends PtMod.Pt of {x:real,y:real}
end
functor Pt3D(M:MyPtSig) = struct
    class My3DPt extends M.MyPt of {z:real}
    fun getZ: My3DPt -> real
    extend fun getZ (My3DPt {x=_,y=_,z=z}) = z
end
```

The `MyPtSig` signature is the interface of a structure containing a `Pt` subclass with x- and y-coordinates. `Pt3D` is a functor that takes any structure meeting the `MyPtSig` signature and returns a new class representing a 3-dimensional version of the given kind of point. In order to safely typecheck the functor body given only the signature of its parameter, we must carefully define the *functor matching* rules defining when a structure can legally be passed to a given functor, balancing the tradeoff between expressiveness and modularity. It may additionally be necessary to restrict some of the idioms in the functor body for safety. For example, allowing the functor body to add arbitrary function cases to existing functions may be hard to typecheck, particularly if the patterns refer to classes in the parameter signature, whose identities are unknown.

3.2.4 Implementation

As part of our implementation of EML, we will develop modular compilation strategies. The MultiJava compilation strategy is a start, although parts of it are specific to the details of the Java virtual machine. Unlike MultiJava, EML doesn't need to run on a fixed virtual machine, so there is more flexibility to generate efficient dispatching code, including using the techniques described in section 3.1.3. ML's propensitity to pass functions as first-class values through the program will require some new techniques beyond the MultiJava implementation strategy. A simple strategy would just represent a function value by a mutable reference cell which can be updated if an `extend fun` declaration is executed, but more sophisticated closure layouts are possible.

A particular goal is for the compiled code to be as efficient as in ML when EML's extra expressiveness is not used. In part to enable more efficient implementation, we will explore the use of extra class and function modifiers that allow the programmer to control the extensibility of their classes and functions. For example, a class could be declared `sealed`, signifying that new subclasses of the class cannot be declared outside of the current module. The `sealed` modifier generalizes the notion of a "final" class in Java. If all the classes in a given hierarchy are sealed, then no new variants can be defined outside of the current module, and all the regular ML datatype layout optimizations can be applied to compactly represent values of the datatype. Similarly, we can declare a function to be `sealed`, signifying that no new function cases can be added outside of the current module, allowing the indirection described above to be removed.

3.3 Case Studies

As part of previous work, an implementation of MultiJava has been completed and is available for public download at http://www.multijava.org. The MultiJava compiler, called *mjc*, compiles MultiJava source code directly to regular Java bytecode, which can then be run on a stock Java virtual machine. MultiJava is completely backward-compatible with Java, so existing Java code can be compiled with mjc. Backward-compatibility greatly decreases the burden of porting one's code to get the benefits of MultiJava, as existing Java code can be ported to MultiJava incrementally.

We will modify the mjc compiler, and write a special Java class loader, to support the Relaxed MultiJava language. We will then translate several large Java programs into Relaxed MultiJava, to assess its utility and practicality. We also will translate some existing programs into EML to study its effectiveness. The rest of this section describes these case studies.

3.3.1 one.world

one.world [Grimm et al. 01a, Grimm et al. 01b] is a platform for pervasive computing, written as a 40,000-line Java program. The one.world platform is *event-based.* Clients of one.world implement *components* that respond to asynchronous *events*. In the one.world framework, components are simply subclasses of the `Component` class, events are subclasses of the `Event` class, and events are handled by implementers of the `EventHandler` interface, which supports an operation `handle` accepting an `Event`. Typically a component class will contain an inner class that implements the `EventHandler` interface, which performs the event dispatching necessary for performing the component's task. For example, a simple logging component might define an event handler that records a particular message in a log file, depending on the kind of event received.

Because one.world is written in Java, clients are forced to perform event dispatching by hand. A prototypical `handle` method has the following form:

```
boolean handle(Event e) {
    if (e instanceof Event1) {
        Event1 ee = (Event1) e;
        ... // specialized behavior for Event1
    }
    ...
    else if (e instanceof EventN) {
        EventN ee = (EventN) e;
        ... // specialized behavior for EventN
    }
}
```

Performing event dispatching in this manner is very tedious and error-prone. Each "dispatch" requires an explicit run-time class test followed by a cast to the appropriate class. If the classes used in the run-time class test and the cast differ, the type system will not catch the error, but a run-time error will occur if that path is ever taken. If some event class is not handled or is handled in multiple ways, these kinds of errors will be hard to debug, manifesting themselves in unpredictable ways at run time, If events form a subtyping hierarchy, care must be taken to perform the class tests in order, from most-specific to least-specific, so that a particular event will be dispatched to its best "handler." For these reasons, undergraduate students building distributed applications in one.world ended up spending most of their time exploring design patterns and various wrapper classes that could make performing event dispatching more palatable.

These problems can be resolved in Relaxed MultiJava, where event handling would naturally be implemented as a collection of multimethods:

```
boolean handle(Event@Event1 e) {
    ... // specialized behavior for Event1
}
...
boolean handle(Event@EventN e) {
    ... // specialized behavior for EventN
}
```

This version allows dispatching to be expressed much more declaratively, obviating the need for the explicit class tests and run-time casts. Relaxed MultiJava will signal a warning at modular compile time if any event is not handled or handled ambiguously. The order of the handlers is irrelevant; event dispatching will be performed in specificity order automatically, using Relaxed MultiJava's multimethod dispatching semantics. In addition, if event handlers form a subtype hierarchy, a given handler can easily inherit relevant multimethods from its superclass, obviating the need to duplicate code.

We will port one.world event handlers to use the style presented above, to explore whether the Relaxed MultiJava type system offers adequate expressiveness in the common case. We also will study other parts of one.world to see how Relaxed MultiJava's features could aid their implementation. We hope to have one.world clients use Relaxed MultiJava when building new applications, to study how Relaxed MultiJava aids in making new applications easier to write, read, and maintain.

3.3.2 Labscape

Labscape [Arnstein et al. 01] is an ubiquitous computing platform for use in a biology laboratory, written in Java. The aim of the platform is to provide a common ground for passing information between the biology workbench, where physical experiments are carried out, and the computer, where digital simulations and data processing are performed. Like one.world, Labscape is structured as an event-based system. For example, a test tube may contain a sensor that causes an event to be triggered in Labscape when the test tube fills beyond a certain point. This event can then be handled by a component as necessary, for example by simply recording the time at which the fill point was reached.

A key requirement of the Labscape platform that distinguishes it from one.world is the ability to dynamically add components, events, and event handlers to the system as it runs. For example, when the biology laboratory obtains a new kind of sensor, the laboratory manager will write a new event type corresponding to the sensor, write any new component applications that take advantage of the new event, and update existing components to be able to correctly handle the new event. It would be very undesirable to be forced to shut down the entire system, modify the code, recompile, and rerun the platform each time the system needed to be extended in this way. Instead, the platform should be able to run continuously, accepting updates "on-the-fly" as they arrive.

Because Java supports dynamic class loading, some aspects of the dynamic-update requirement can already be handled naturally. For example, it is relatively straightforward for a new event or component class to be dynamically loaded. However, updating existing code is much more difficult, requiring advance planning and unnatural encodings. As mentioned above, existing event handlers must be able to be updated to handle new events as they enter the system. In the current implementation of Labscape, therefore, an event handler is represented as a list of objects, each with an "apply" method that handles a particular kind of event. Event dispatching consists of walking down the list of handler objects, using run-time class tests to determine if each handler is applicable to the given event, invoking the first one that applies. The event handler can be updated dynamically by adding a new object to the list. This approach is even more tedious, error-prone, and type-unsafe than the approach used in one.world.

In porting Labscape to Relaxed MultiJava, a key goal is to be able to model an event handler as a collection of multimethods, as described in the previous subsection. On top of the advantages described in the previous subsection, this approach allows Labscape's event handlers to be expressed in a way that is much closer to the logical view of an event handler. In addition, the Relaxed MultiJava approach is potentially much more efficient, obviating the need for the wrapper objects around each handler case. The new features of Relaxed MultiJava's relaxed type system can be exploited, allowing new `handle` methods to be defined in new compilation units that are dynamically loaded into the running application. When these compilation units are compiled, a warning will signal the possibility for violations of type safety. When the compilation units are eventually loaded dynamically, the specialized class loader will verify the safety of any new methods.

3.3.3 UrbanSim

UrbanSim [Waddell et al. 00] is a software-based simulation model designed to be used for integrated planning and analysis of urban development, incorporating the interactions between land use, transportation, and public policy. It is intended for use by metropolitan planning organizations and others needing to interface existing travel models with new land use forecasting and analysis capabilities. The simulation engine is 300,000 lines of Java source code, making it one of the largest freely available Java programs today. UrbanSim needs to be extensible, to support the easy addition of new kinds of simulation agents. We will investigate how Relaxed MultiJava's features can support extensibility along multiple dimensions.

3.3.4 EML Studies

To assess EML, we will find existing, interesting ML and O'Caml (the main implementation of Objective ML) programs, and see how their datatypes, functions, and (for O'Caml) classes can be re-expressed using EML classes and extensible functions. We also plan to translate key parts of our large body of Cecil programs into EML. Cecil uses a global typechecking strategy and can therefore safely allow arbitrary multimethod idioms. It will be interesting to study whether EML is expressive enough to support common Cecil idioms, despite EML's restrictions for modular type safety.

3.4 Beyond MultiJava and EML

Although MultiJava and EML broaden the abilities of the OO and functional paradigms to support modular extensibility, there are several useful idioms that are not addressed by these languages. For example, while MultiJava supports the addition of new methods to existing classes in a modular way, it does not provide support for updating existing methods. If a method requires updating, it must either be modified in place, or a subclass must be created that contains an overriding method with the necessary behavior. In the latter case, existing code must still be modified, as existing clients of the original abstraction must become clients of the new subclass.

Several technologies have emerged in recent years to address finer-grained extensibility issues, including the external augmentation of existing methods. For example, as described in section 2.1, AspectJ's "before" and "after" methods allow a form of method combination. AspectJ also includes an "around" construct that allows a method to be completely redefined externally, with control over whether and when the original method gets invoked. Hyper/J allows almost arbitrary fragments of classes, called *hyperslices*, to be created and combined in flexible ways to form complete classes. Hyperslice combination is directed by a set of composition rules. For example, two hyperslices containing methods of the same name could be combined by having one of the methods shadow the other or by concatenating the method bodies into a single method. Hyper/J also supports *on-demand remodularization,* whereby existing classes may be decomposed into hyperslices (including overlapping hyperslices) after the fact, which can be composed to form variants on the original class.

While these kinds of techniques are quite flexible, allowing almost arbitrary modifications and extensions to existing abstractions without modifying existing code, they currently come at the expense of modular reasoning. Aspects in AspectJ cannot be safely typechecked and compiled modularly. Instead, a global preprocessing phase rewrites an AspectJ program into an ordinary Java program, and subsequently ordinary Java typechecking and compilation is performed. This means that an aspect cannot be guaranteed to be typesafe until the entire context in which it is placed is available. As a simple example, two aspects may add the identical method to an existing class, so the clash can only be detected when the class and all associated aspects are available. Similarly, hyperslices in Hyper/J are not typechecked and compiled modularly; regular Java typechecking and compilation occurs after hyperslice combination is complete.

We propose to explore extensions to these languages that support finer-grained extensibility à la AspectJ and Hyper/J, while preserving safe modular reasoning. To this end, we will develop richer interfaces for components that are capable of expressing the necessary requirements and assurances of a component. We can build on MultiJava's technique for handling the modular addition of new methods to existing classes to handle the example described in the previous paragraph. However, the interface of an aspect may also require information that goes beyond traditional "type" information, in order to define the legal compositions. For example, an "after" method may rely on a specific invariant of the original method. However, an independent aspect can create an "after" method that violates this invariant. The correctness of the final method produced therefore depends on the order in which the two "after" methods are combined. If the first method's requirement is made part of its interface, then we can ensure that the method can only be combined with other methods in such a way as to maintain the invariant. Similar problems arise in the context of Hyper/J. In that case, each hyperslice is guaranteed to be *declaratively complete*, specifying the types of the identifiers it relies on from other hyperslices to be eventually combined with. However, this type information is not enough to reason modularly about the ways in which the given hyperslice can be safely combined with other hyperslices to form a complete class.

It will also be interesting to explore techniques for reconciling extensibility with typical forms of encapsulation as found in OO languages. For example, aspects in AspectJ have access to the private fields and methods of the classes they extend. Although this privilege allows aspects to be quite flexible, it essentially allows any client to obtain access to private information simply by writing a new aspect. In MultiJava, on the other hand, top-level methods only have as much privilege as is allowed by the associated compilation unit. In particular, private data can only be accessed if the top-level method is defined in the same compilation unit as its class. It would be useful to define a middle ground that differentiates between clients that require privileged access and those that do not, with associated responsibilities for each kind of client.

Finally, combining aspect- and subject-oriented technologies with EML's parametric polymorphism and functors can lead to sophisticated tools for fine-grained reuse. For example, a parametrically typed aspect could be instantiated for use in multiple typing contexts. With functors, an aspect can be parametrized by the class which it extends, providing a powerful and reusable abstraction for easily extending an existing class in some way.

4 Contributions and Impact

Our long-term goal is to develop mechanisms for constructing reusable, reliable software components. We believe that providing language support for modular, multidimensional extensibility is an important step toward this goal. Multidimensional extensibility allows a component to be reused and extended in many ways which were not anticipated by the original implementer of the component. Modular type safety provides these components and their flexible extensions a measure of reliability, guaranteeing that run-time type errors cannot occur, regardless of the context in which the components and extensions are eventually placed. Richer interfaces further increase reliability by strengthening the modular guarantees provided by the language. Through this project, we hope to develop the technology to build language components that support the modular addition of new variants, new operations, and even new statements, using a general predicate-based dispatching model and flexible method combination rules, all with modular typechecking and compilation. This technology will remove the barriers to wider use of fine-grained extensibility, as currently embodied in the aspect- and subject-oriented paradigms, whose utility is compromised by a lack of modular reasoning.

We will make our research results available both to our own students and to the outside research community, not only through papers but also through implementations of Relaxed MultiJava and EML. Feedback from the case studies and from others in the outside community on the practical strengths and weaknesses of our work will be critical in understanding the extensibility needs of real systems and will lead to iterative refinements and improvements of the languages. We also plan to use MultiJava and EML as teaching languages in programming language classes, creating a captive user community from which to solicit early feedback, and also exposing students to current research in programming language design.

5 Time Schedule

This section outlines our planned schedule for completing the proposed research.

Year 1

- We will design and implement Relaxed MultiJava, including incremental typechecking algorithms that integrate with Java's lazy class loading.
- We will study the effectiveness of Relaxed MultiJava in aiding development and maintenance of applications written in one.world and Labscape.
- We will begin the design the EML language and type system, including treatment of patterns, parametric polymorphism, and functors.

Year 2

- We will develop strategies for dynamic generation of efficient method dispatchers, and incorporate them into the Relaxed MultiJava implementation.
- We will study the effectiveness of Relaxed MultiJava in the context of the UrbanSiim project.
- We will make an initial public release of the Relaxed MultiJava implementation.
- We will complete the design of the EML language, and prove it sound. We will begin work on the EML implementation.

Year 3

- We will complete the implementation of EML.
- We will translate at least parts of existing ML, O'Caml, and Cecil programs into EML.
- We will make an initial public release of EML.
- We will extend Relaxed MultiJava to support some of the features of aspect- and subject-oriented programming in a modular, type-safe setting.

6 Summary of Results from Last Five Years of Prior NSF Support

In the last five years, the following NSF grants have supported our work:

grant #	title	duration	amount
CCR-9457767	National Young Investigator Award	1994-1999	$312,500
CCR-9503741	Analysis and Optimization of Object-Oriented Languages	1995-1998	$149,406
CCR-9970986	Towards Practical, Expressive Multimethods	1999-2002	$297,620
CCR-9975057	A Staged Compilation Architecture for Program Optimization	1999-2001	$500,000
CCR-0073379	Representation Specification and Optimization of Object-Oriented Languages	2000-2003	$249,782

Cecil: A Purely Object-Oriented Language
MultiJava: An Extension of Java with Open Classes and Multimethods

The first and third NSF grants listed above supported our research on the Cecil language design project and follow-on language design work. Cecil is a high-level, purely object-oriented language [Chambers 93a, Chambers 96], used as a vehicle to investigate language design issues in classless object models, multi-methods, modules and encapsulation, and static type systems. New technical results include reconciling multi-methods with object-oriented notions [Chambers 92], predicate classes [Chambers 93b], efficient type systems and module systems for languages with multi-methods [Chambers & Leavens 94, Chambers & Leavens 95b], modular type checking for languages with multi-methods and open classes [Millstein & Chambers 99, Clifton et al. 00, Millstein & Chambers], flexible polymorphic static type systems [Litvinov & Chambers 98, Litvinov 98], and a general dispatching model unifying and extending multi-methods, predicate classes, and pattern-matching [Ernst et al. 98a]. The Vortex optimizing compiler (described below) is written entirely in Cecil, providing a realistic test-case for evaluating the language. The Cecil front-end for Vortex was released to the external research community along with the rest of Vortex in April 1997 and then again (along with a stand-alone Cecil interpreter) in February 1999.

We have built on the Cecil work by adapting the essential ideas to other, more-mainstream languages. For example, the MultiJava language extends Java with open classes and multi-methods, while retaining Java's modular typechecking.and compilation strategy [Clifton et al. 00]. A prototype MultiJava compiler was released to the outside research community in summer 2001 and is currently being used at Iowa State University as the platform for implementing the Java Modeling Language, a behavioral interface specification language for Java. More details on MultiJava are found in Sections 2.1 and 3.1.1.

Vassily Litvinov is nearing completion of his Ph.D., having worked on both the Cecil and Vortex projects, and Todd Millstein is closing in fast behind, having worked on the Cecil and MultiJava projects. Claudia Chiang, Stuart Williams, Jean Kaiser, and Lauren Bricker received Master's degrees based on early work on the Cecil project. Several undergraduates worked as researchers on the Cecil project, including Vitaly Shmatikov (now at Stanford) and Phiem Huynh Ngoc. Two undergraduates are currently working on the MultiJava project.

Vortex: Optimizing Compilation for Object-Oriented Languages

The first three grants list above supported our work on the Vortex optimizing compiler project. The Vortex project has been very successful, producing new technical results in automatic inlining [Dean & Chambers 94], selective recompilation [Chambers et al. 95], link-time class hierarchy analysis [Dean et al. 95b], profile- and class-hierarchy-guided selective procedure specialization [Dean et al. 94, Dean et al. 95a], profile-guided class prediction [Grove et al. 95], interprocedural class analysis frameworks and fast interprocedural class analysis algorithms [Grove et al. 97, DeFouw et al. 98, Grove & Chambers], efficient implementation of advanced method dispatching schemes [Chambers & Chen 99], frameworks for intra- and interprocedural dataflow analysis [Chambers et al. 96], elimination of unnecessary synchronization in Java programs [Aldrich et al. 99, Aldrich et al.], and measurements of optimization effectiveness across languages [Dean et al. 96]. A common theme in these works is assessment of the effectiveness of the optimization techniques on substantial, realistic benchmarks, typically on the order of 10,000-50,000 lines in size, which often are an order of magnitude bigger and more complex than benchmarks studied by previous work. Overall, the techniques studied in Vortex yield speed-ups for the large benchmark programs of 30-40% for C++, of 50-80% for Java (over straightforward translation to C and subsequent optimized compilation of the C code, which is a factor of 4 better than interpreted Java and a factor of 2 better than typical Java "just-in-time" dynamic compilers), and a factor of more than 7 for purely object-oriented Cecil programs. The Vortex compiler system has been in daily use internally for its own development since early 1994, helping to ensure that the techniques used in Vortex are practical on a normal program development environment. The entire Vortex system, including front-ends for C++, Java, and Cecil, and back-ends for portable C code and SPARC assembly code, was originally released to the outside research community in April 1997; a second major public release took place in February 1999.

Many students received training in compiler research as part of the Vortex project. Jeffrey Dean received his Ph.D. in December 1996, and David Grove received his Ph.D. in October 1998. Charles Garrett received his Master's degree in March 1994. Several undergraduates worked as researchers on the Vortex project, including Greg DeFouw (later at Stanford), Tina Wong (later at U.C. Berkeley), Ben Teitelbaum (later at U. Wisconsin), MaryAnn Joy, Brian O'Neill, Leo Shum, and Matthew Parker. Vortex also has been used as the infrastructure for implementation projects in my graduate compilers class for several years, aiding the education of dozens of graduate students. Vortex has also been used as an experimental infrastructure for research groups outside the University of Washington.

Whirlwind: A Next-Generation Optimizing Compiler for Object-Oriented Languages

The last two grants listed above supported our research on the new Whirlwind compiler project. Whirlwind builds on the lessons and experience gained with the Vortex project. One major feature in Whirlwind is the use of a sparse intermediate representation, and a framework for automatically composing modular analyses and transformations to achieve better optimization results [Lerner et al. 02]. Another major research direction is the development of techniques for describing the layouts of data structures, for transforming the layouts into more optimized forms [Dolby 97, Dolby & Chien 98, Dolby & Chien 00], and for checking that compiler optimizations preserve important invariants of the code that manipulates these data structures. Several Ph.D. students are currently working on the Whirlwind project.

DyC: Staged Run-time Compilation for C
Calpa: Automatic Staged Run-time Compilation for C
SCF: A Staged Compilation Framework

The first and fourth grants listed above also supported work on the DyC run-time compilation system for C [Auslander et al. 96, Grant et al. 97, Grant et al. 99, Grant et al. 00a, Grant et al. 00b]. The DyC project has developed effective, practical techniques for performing selective run-time compilation on standard imperative programs, driven by declarative and expressive user annotations. A key contribution of the DyC project is its approach to staging several run-time optimizations, with static preplanning drastically reducing the cost of the run-time compilation stage. DyC has been evaluated on larger benchmarks than related research projects. These studies showed that dynamic compilation could produce speedups by generating code that was of sufficient quality to pay back the costs of compiling during program execution, and that the payoff occurred at input sizes that were within the norm for the particular application.

Two current projects are building on our experience with DyC. The Calpa project strives to automate the selection of where and on what to dynamically compile, combining static analysis and profile feedback with a cost/benefit model of DyC's optimizations and the target machine [Mock et al. 99, Mock et al. 00]. The SCF project has built an initial staged compilation framework that can automatically convert a regular unstaged pipeline of optimizations into a staged one, given information about the range of programs to be compiled in the later stages [Philipose et al. 02].

Brian Grant received his Ph.D. based on his work on the DyC project. Markus Mock is nearing completion of his Ph.D. based on his DyC and Calpa work, and Matthai Philipose is also nearing completion of his Ph.D. based on his DyC and SCF work.