Towards Safe Modular Extensible Objects

Craig Chambers and Gary T. Leavens
TR #94-17a
August 1994, revised September 1994

To appear in the proceedings of the *OOPSLA '94 Workshop on Subjectivity in Object-Oriented Systems*, October 1994.

Keywords: Multi-methods, object-oriented programming, subjectivity, encapsulation, modules, packages, Cecil language.

1994 CR Categories: D.3.2 [*Programming Languages*] Language Classifications — Object-oriented languages; D.3.3 [*Programming Language*] Language Constructs and Features — Modules, packages.

Department of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, Iowa 50011-1040, USA

Towards Safe Modular Extensible Objects

Craig Chambers

Department of Computer Science and Engineering
309 Sieg Hall, FR-35
University of Washington
Seattle, Washington 98195
(206) 685-2094; fax: (206) 543-2969
chambers@cs.washington.edu

Gary T. Leavens

Department of Computer Science
229 Atanasoff Hall
Iowa State University
Ames, Iowa 50011-1040
(515) 294-1580
leavens@cs.iastate.edu

Abstract

We discuss the module system of the Cecil language, which has a flexible means of customizing views of objects. Multi-methods with invocation based on static scoping, a module system, and extension modules are used to allow object extension while preserving type safety.

1 Introduction

In traditional object-oriented languages, an object is defined as a single monolithic unit. An object exports exactly one interface to its clients, specified when the object was defined. A common design issue encountered in such languages is whether to specify an operation as a method of the object's class (complicating the object's interface) or as an external operation that takes the object as an argument (sacrificing the ability to do dynamic dispatching and to gain direct access to the object's representation). Defining specialized subclasses of an existing class is not always an acceptable alternative, since subclassing does not allow existing objects to be extended with specialized behavior nor does it support sharing of objects that have different capabilities when viewed by different clients.

In the Cecil language [Chambers 92, Chambers 93] we are exploring a more flexible alternative that allows each individual client (embodied in the program by a scope, which is usually a module or group of nested modules) to customize its view of an object [Chambers & Leavens 94]. Each module can add new multi-methods, instance variables, and even inheritance and subtyping relationships to objects defined in lexically enclosing or imported modules. Moreover, extensions inside a module are encapsulated so that modifications to the shared objects are not visible to other modules and thus do not affect the general interface of the object as seen by other modules.

For example, the standard `String` class could be defined in the standard library module with a minimal interface. Individual client modules of `String` could then augment this interface with their own specialized behavior. For instance, a text editor application could add a tab-expansion method to the `String` class, inside the `TextEditor` module. Within the `TextEditor` module, its extensions to the `String` class are first-class operations of the `String` class; there is no difference between calling the tab-expansion method and a built-in method that applies to strings. However, by making such extensions inside modules, other modules do not have their view of the `String` class polluted by the text editor's specialized extensions, and vice versa.

Our original motivation for studying extensible objects was our work on modular typechecking of multi-methods. A multi-method is a method that can dispatch on the dynamic class of any of its arguments, not just the first receiver argument. When defining a method that dispatches on two or more arguments, potentially of different classes, the method most naturally is viewed as an extension of the objects on which

it dispatches. By placing multi-methods inside modules, they can be isolated from multi-methods in other modules and typechecked independently. One complexity of multi-methods not shared by other kinds of object extensions is that modules containing multi-methods that are correct in isolation can fail when combined in a single program. For instance, two independently-developed multi-methods could both be applicable to handle some message, e.g. "+"(complex,num) and "+"(num,fraction) when adding a complex number and a fraction, but neither multi-method would override the other, thereby leading to a ambiguous message lookup error. Our module system includes a well-formedness requirement on the modules included in a program to ensure that such ambiguities are detected statically.

Given this intuitive idea of the potential advantages and pitfalls of this more flexible object model, our current work is directed towards understanding its precise semantics. In particular, the semantics of method lookup needs to consider not only the dynamic classes of the message's arguments, but also the module from which the message is being sent. Additionally, we are investigating static type systems that can guarantee that no message lookup errors will occur at run-time. An initial informal semantics and a typechecking algorithm for languages containing modules and multi-methods is being presented in the main OOPSLA conference [Chambers & Leavens 94]. We are currently continuing this work to address fully extensible objects, to formally specify the run-time semantics and the static typechecking rules, and to prove that the static typechecking restrictions are sufficient to guarantee that no type errors occur at run-time.

Acknowledgments

Chambers's work is supported in part by a National Science Foundation Research Initiation Award (contract number CCR-9210990) and several gifts from Sun Microsystems. Leavens's work is supported in part by a National Science Foundation grant (contract number CCR-9108654).

More information on the Cecil language and implementation project can be found through the World Wide Web under http://www.cs.washington.edu/research/projects/cecil/www/cecil-home.html and via anonymous `ftp` from cs.washington.edu:/pub/chambers.

References

[Chambers 92] Craig Chambers. Object-Oriented Multi-Methods in Cecil. In *ECOOP '92 Conference Proceedings*, pp. 33-56, Utrecht, the Netherlands, June/July, 1992. Published as *Lecture Notes in Computer Science 615*, Springer-Verlag, Berlin, 1992.

[Chambers 93] Craig Chambers. The Cecil Language: Specification and Rationale. Technical report #93-03-05, Department of Computer Science and Engineering, University of Washington, March, 1993.

[Chambers & Leavens 94] Craig Chambers and Gary T. Leavens. Typechecking and Modules for Multi-Methods. To appear in Proc. *OOPSLA '94*, ACM. (The extended version of this paper is TR94-03a, Department of Computer Science, Iowa State University, Ames, IA, August 1994. It can be obtained by e-mail from almanac@cs.iastate.edu or by anonymous `ftp` to ftp.cs.iastate.edu in directory pub/techreports/TR94-03)