

# Compilation Strategies as Objects

Anurag Mendhekar, Gregor J. Kiczales and John Lamping

Published in Proceedings of the 1994 OOPSLA Workshop on Object-Oriented Compilation -- What are the Objects? 1994.

© Copyright 1994 Association for Computing Machinery, Inc.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Publications Dept, ACM Inc., fax +1 (212) 869-0481, or [permissions@acm.org](mailto:permissions@acm.org).

*Appears in the proceedings of the OOPSLA'94 Workshop on OO Compilation.*

# Compilation Strategies as Objects

Anurag Mendhekar

*Indiana University*

Gregor Kiczales, John Lamping

*Xerox PARC\**

©1994 Xerox Corporation.

## Abstract

In this paper we present an overview of the metaobject protocol approach to compilation. We take the position that object orientation in a compiler can be put to effective use in opening up the compiler for modification by the user. Interestingly, the natural scopes of effect of user intervention don't respect syntactic boundaries, so we require objects that are not just elements of the abstract syntax tree. We introduce a new kind of intermediate object for the process of compilation so that user modification of compilation strategies can be carried out in a coherent manner. We give some examples of user customizations, and outline the architecture of our Scheme compiler based on these principles.

## 1 Introduction

We have designed a user customizable Scheme compiler. It is object-oriented and like many object oriented compilers, our compiler creates an object (a metaobject in our terminology)<sup>1</sup> for each piece of syntax in the program to be compiled. Users can customize the compilation process by annotating source level expressions with indications of alternative classes to use for their metaobjects, which lead to different compilation strategies for those objects.

But user customizability raises the spectre of inconsistency. If a user can change how parts of a program are compiled, how can they be assured that the assumptions made in compiling different parts of their programs will be consistent? For example, if a procedure is compiled with a different, special purpose, calling convention, how

---

\*3333 Coyote Hill Rd., Palo Alto, CA 94304; anurag@cs.indiana.edu, {gregor,lamping}@parc.xerox.com.

<sup>1</sup>We use the term metaobject for objects the compiler manipulates—either parts of the abstract syntax tree or the new contract metaobjects this paper presents. They are “meta” in the sense that they are about the program rather than objects the program manipulates.

can the user be assured that all call sites to the procedure will use that convention? Another way of putting it is that compile time customization can't be done at the level of individual parts of the program; it must be coordinated across interacting parts.

We have introduced a new kind of compile-time metaobject to address these issues. Each of these new metaobjects controls the compilation of all uses and creations of a particular data structure. These metaobjects correspond to contracts between closely interacting parts of a program. They occupy an intermediate level of granularity between the fine-grained level of syntactic structure and the coarse-grained level of the program as a whole.

The bulk of this paper focuses on this issue. We first set the stage by giving some motivation for the kind of customizations that our compiler supports. We then describe the new level of granularity for resolving the consistency issue and present its manifestations as the intermediate compile-time metaobjects which are representative of the compilation strategy. With this intermediate stage in the background, we describe the overall structure of the compiler and how the customization is actually carried out.

## 2 Motivation

Consider a world in which you get to have your cake and eat it too. You do your day-to-day programming in a high-level language like Lisp, that provides you with powerful abstractions, but on those crucial occasions where its important that an abstraction be implemented in a particular way or that a slightly different abstraction be available, you get to help the compiler achieve that end, in a principled and modular way. In this world, you get to enjoy the benefits of high-level abstraction without paying traditional performance, interoperability and functionality costs. Our goal is to move toward this world by opening the implementation of a compiler.

There have been several efforts, such as [KdRB91, WY88, IO91], to open the implementation of programming languages, using techniques such as reflection and metaobject protocols, to give users the ability to examine and affect the implementation of their program. These efforts have provided introspective capabilities, enhanced expressive power, and improved implementation efficiency. But, with the exception of [Rod92], they have primarily focused on user access to the implementation at run time. We believe that access to compile time implementation decisions may be equally valuable. In particular, we hope to achieve very efficient implementations of high level languages by giving users wide control over how their program is implemented in terms of lower level data structures and operations, while having the overhead that comes with providing user access be limited to compile time.

We have used the customizability of our compiler to implement many examples (at user-level), some of which are:

- Alternative representations of data-types such as arrays. We have implemented array structures commonly used in scientific programming that use space and cache resources more efficiently e.g., triangular arrays, arrays with either row-major or column-major layout, blocked layouts etc.
- In-lined procedures and loops. Loops which are commonly expressed in Scheme as recursive procedures can be implemented more efficiently when they are known to be loops.

- Procedures that dispatch on the number of arguments they receive — such as those returned by the case-lambda facility of Chez Scheme [Dyb91].
- Procedures that have extra data associated with them. Functionality like this is what underlies objects in T [RA82, RAM84], entities and application hooks in MIT Scheme [Han91] and funcallable instances in CLOS [BDG<sup>+</sup>88, KdRB91].
- Data abstractions that are truly opaque, immutable, or only mutable by privileged parts of a program (e.g. a record which cannot be mutated once it is initialized).
- Powerful, seamless access to code and data in the surrounding environment (i.e. Unix, DOS, Windows).
- Powerful analysis of programs to determine, both statically and dynamically, information such as where a certain value can reach, where the value that reaches a certain point could have come from, whether there are dead branches in the code, etc.

We demonstrate a simple but powerful example in which we extend one of the language's basic structures, procedures, without affecting performance, reliability, or the user's having to touch the source code of the compiler. This is the example of procedures with an extra data cell. We will introduce new syntax for such procedure with a new keyword, `Dlambda`, and we will introduce two new functions, `procedure-data` and `set-procedure-data!` to access the data cell of such procedures.

To link these new facilities with the necessary compilation routines, we need to have them engender abstract syntax tree metaobjects that are instances of new classes. Our compiler uses a special annotation syntax with curly braces to indicate that the abstract syntax tree metaobject that represents the immediately following expression has a special class. So we will define a macro `Dlambda` that expands to:

```
{procedure-with-data}(lambda (...) ...)
```

where the ellipses denote the obvious syntactic entities from the input to the macro, and we will define the new procedures as follows:

```
(define procedure-data
  {procedure-with-data-reader}dummy-reader)

(define set-procedure-data!
  {procedure-with-data-writer}dummy-writer)
```

So now we have enough information in the program text to indicate all uses of the new feature. Next, the designer of this facility has to define the three new classes so that their constructs will be compiled correctly. We won't go into the details of the compilation protocol, but will focus on the most crucial issue: the compilation of other parts of the program may have to change as well. For example, calls to a `procedure-with-data` may have to be compiled differently. Similarly, if `procedure-data` might get an ordinary procedure, it will have to be compiled to do a run-time check, or the original procedure creation will have to be made to create a `procedure-with-data`.

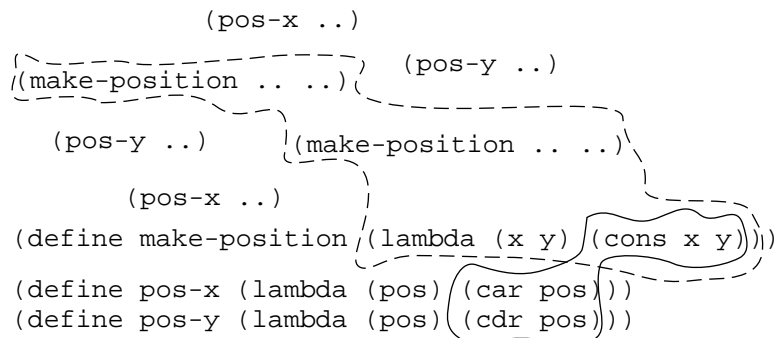


Figure 1: An illustration of how different parts of the program must be compiled consistently.

### 3 Compile-time Metaobjects

As this example shows, how one part of a program is compiled also affects how another part of the program may be compiled. For another example, the layout of procedures determines how they are going to be applied. The decision about layouts thus not only affects the compilation of procedures but also applications. The calling convention decides how variables in the body of a procedure are accessed. The decision about whether to pass arguments in registers or on the stack thus determines how variable accesses are to be compiled. The reverse of both these examples is also true. If a decision is made to access variables only from registers, the calling convention must be suitably adapted.

Traditional compilers fix the strategy for compiling each language construct, and can thus guarantee consistency by choosing compatible strategies. But our goal is a customizable compiler where different instances of the same construct can be compiled with different strategies. We still want to guarantee consistency, and to make it easy for the user to express a far-ranging effect with a localized annotation.

We have found that most compilation strategy decisions can be characterized in terms how various data structures are represented. Let us consider the kinds of values in a typical program: closures, pairs, vectors, strings, symbols etc.<sup>2</sup> The chosen representation of these values dictates how they are to be manipulated—how they are constructed and how they are destructured.

Imagine now that such a strategy is encapsulated in an object. Let us illustrate this with the example of pairs. A pair strategy is an an object that responds to three messages: `compile-cons`, `compile-car` and `compile-cdr`. Let us call this object a pair metaobject. Sending a `compiler-cons` message to the pair metaobject generates code to construct a new cons cell. Sending a `compiler-car` (`cdr`) message to the pair metaobject along with a compile-time locator of a the cell that will be so constructed, generates code to access the first (second) element of the cons cell. Thus if we had to compile a primitive cons expressions, we simply send a message to the pair

<sup>2</sup>In Scheme, most of these values are first class. In some other languages, however, they aren't. This does not mean that these values don't exist behind the scenes. Concrete representations for these exist and are manipulated similarly.

compile-time metaobject and have it generate the corresponding code. We call these metaobjects *contracts* since they represent an agreement among a set of expressions about implementation strategy.

The equivalent of having a global compilation strategy is that there is exactly one contract for each kind of value. If we are to respect to desideratum of localized specialization of compilation strategy, it is clear that there must be many such contracts for a given kind of value and expressions which use a particular compilation strategy must know exactly the metaobject to use. Consider the following example:

```
(let ((y (cons 99 101)))
  (let ((x (cons 10 12)))
    (+ (car y) (cdr x))))
```

If we were to change the implementation strategy of the `cons` expression binding `y`, the corresponding implementation of the `car` expression must also change, but not those of the of the `cons` expression binding `x` or the `cdr` expression. The `cons` expression for `y` and the `car` expression must now share a compilation contract. Similarly, the `cons` expression for `x` and the `cdr` expression must share another compilation contract. Figure 1 shows another example. Expressions sharing a contract are grouped together diagrammatically. The procedure `make-position` and its applications belong to one procedure contract, and `(cons x y)`, `(car pos)` and `(cdr pos)` belong to the same pair contract.

Contracts, in general, cover forms that might construct or access (i.e. destruct) the same piece of data. Lambdas and applications are one such case: lambdas construct procedures, and applications access the data in the procedure to perform the application. Similarly, `cons` constructs pairs and `car` and `cdr` destruct them. Table 1 lists the different kinds of contracts, with their constructors and destructors. The class of the contract can be influenced by any of its contractees. The contract determines what it is responsible for storing, accumulating requirements from all the participants in the contract. Then it decides how that information will be laid out. For example, the default pair contract will probably decide that it must store two elements, in a two-word structure.

## 4 Structure of the compiler

The compiler accepts abstract syntax tree objects as its input<sup>3</sup>. This abstract syntax tree is analogous to the parse tree of the program but has as nodes objects which are instances of default classes corresponding to kinds of expressions in Scheme. For example, lambda expressions are instances of the class `<lambda-expr>`. Specialized classes can be used for program graph metaobjects to affect their behavior, or the behavior of contract metaobjects derived from them. (We have built a special object-oriented language that facilitates the combination of influences from abstract syntax tree metaobjects upon contract metaobjects See [Kic93] for details.) After the initial parsing and name resolution, data flow analysis is carried out. This information guides

---

<sup>3</sup>This is important because the metaobject protocol is defined on these objects—not on their textual manifestations.

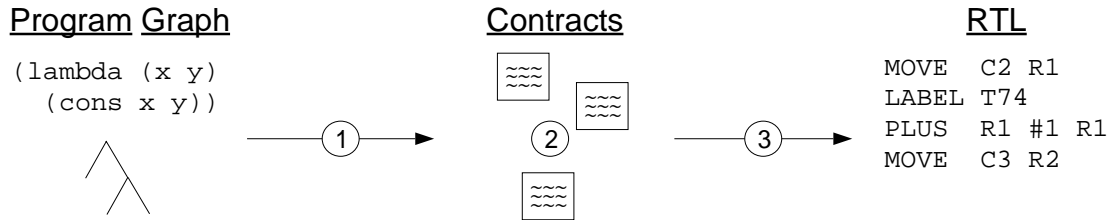


Figure 2: Contracts are an intermediate stage in the process of compilation.

Type	Constructors	Destructors
Procedure	lambda	applications
Pair	cons	car, cdr
Environment	lambda, let, let* and letrec	variable accesses and set!'s
Number	+, -, * ...	+, -, * ...
Vector	make-vector	vector-ref, vector-set!
⋮	⋮	⋮

Table 1: A few of the different types of contract, together with the constructors and destructors of the data they govern. In addition, all the various predicates (eq?, pair?, procedure? etc.) can be destructors for any data type.

contract construction so that each expression in the program can be given a contract which will be used to compile that expression. The following sections describe the stages in some detail. Figure 2 gives the over-view of the structure.

## 4.1 Flow Analysis

This phase of the compilation builds a contract metaobject for each group of abstract syntax tree metaobjects that must agree on runtime data they share. Each group is made disjoint from all the others and as small as it can be while guaranteeing that metaobjects that must agree will be in the same group.

The information used in grouping the abstract syntax tree metaobjects under contracts is which constructors' results could reach which destructors, which is calculated by flow analysis. In the case of lambda and applications this information is the familiar call graph. For more details see [LKRR92].

Since any flow analysis of Scheme is inherently imprecise, our default analyzer can lead to contracts that cover more abstract syntax tree metaobjects than necessary, which means coarser units of decision making. Our analyzer sticks to the MOP philosophy; it follows an extensible protocol so that the user can augment it with declarations or even metacode necessary to result in any smaller contracts the user wants. The details of this protocol are beyond the scope of this paper.

## 4.2 RTL Generation

For reasons of portability, our MOP generates machine-independent RTL that is later translated into native binary in a non-specializable way. RTL generation is done by a collaboration between the abstract syntax tree and contract metaobjects. This collaboration can be illustrated by the generation of the code for an application. When the application is requested to generate its code, it requests its contract metaobject to generate a call, passing it the abstract syntax tree metaobjects for the procedure and for each of the arguments. The contract metaobject requests each of those abstract syntax tree metaobjects to generate code for their expressions, placing the results in the places required by the call contract. Finally, the contract metaobject generates the remaining code for the application to do frame management and the actual calling.

## 5 Related Work

Metaobject protocols are a synthesis of procedural reflection and object-oriented techniques and have been developed by a number of researchers. [Kic92] presents a recent summary of the ideas in this work. But the idea that users should have meta-level control over the programming language goes back much farther. Compilers have supported declarations ranging from the very general, such as whether to optimize for space or for speed, to the more specific, such as whether a given procedure should be inlined. Some of these facilities have been quite powerful, such as those found in the SETL representation sublanguage [D<sup>+</sup>79]. An early summary of the motivations for providing this access can be found in [SW80].

## 6 Summary

We were motivated to build an object oriented compiler to allow for user control of the compilation process to customize both implementation and semantics. The natural granularity of customizability for compilation is not a unit of program text, but rather a group of data values and the program locations that will interact with them. We create a new kind of compile-time metaobject, the compilation contract, that corresponds to such units, and which allows user adjustments to be expressed locally yet have their consequences take effect consistently.

## Acknowledgments

Over the past several years, we've worked with a number of people in the development of this work. This group has included: Hal Abelson, J. Michael Ashley, Alan Bawden, Jim des Rivières, Mike Dixon, Luis Rodriguez, Erik Ruf, Brian Smith and Amin Vahdat. We'd also like to thank our colleagues in the reflection and programming languages community, who have given us much useful feedback on this work: Craig Chambers, Shigeru Chiba, Dan Friedman, Chris Haynes, Satoshi Matsuoka, Mario Tokoro and Akinori Yonezawa.

## References

- [BDG<sup>+</sup>88] Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales, and David A. Moon. Common lisp object system specification. *Sigplan Notices*, 23(Special Issue), September 1988.
- [D<sup>+</sup>79] Robert B. K. Dewar et al. Programming by refinement, as exemplified by the SETL representation sublanguage. *ACM Transactions on Programming Languages and Systems*, 1(1):27–49, July 1979.
- [Dyb91] R. Kent Dybvig. *Chez Scheme System Manual, Revision 2.2, 1991*. Cadence Research Systems, Bloomington, Indiana, 1991.
- [Han91] Chris Hanson. Mit scheme reference manual. Technical Report Edition 1.1, Massachusetts Institute of Technology, November 1991. for Scheme Release 7.1.3.
- [IO91] Yutaka Ishikawa and Hideaki Okamura. A new reflective architecture: AL-1 approach. In *Proceedings of the OOPSLA Workshop on Reflection and Metalevel Architectures in Object-Oriented Programming*, 1991.
- [KdRB91] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.
- [Kic92] Gregor Kiczales. Towards a new model of abstraction in software engineering. In Akinori Yonezawa and Brian Cantwell Smith, editors, *Proceedings of the International Workshop on New Models for Software Architecture '92; Reflection and Meta-Level Architecture*, pages 1–11, Tokyo, Japan, November 1992.
- [Kic93] Gregor Kiczales. Traces (a cut at the “make isn’t generic” problem). In Shojiro Nishio and Akinori Yonezawa, editors, *Proceedings of International Symposium on Object Technologies for Advanced Software (ISOTAS'93)*, pages 27–43. JSST, Springer-Verlag, 1993. Lecture Notes in Computer Science 742.
- [LKRR92] John Lamping, Gregor Kiczales, Luis H. Rodriguez Jr., and Erik Ruf. An architecture for an open compiler. In *Proceedings of the IMSA '92 Workshop on Reflection and Meta-level Architectures*, 1992. Also to appear in forthcoming PARC Technical Report.
- [RA82] Jonathan A. Rees and Norman I. Adams. T: a dialect of lisp or, lambda: the ultimate software tool. In *Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming*, pages 114–122, 1982.
- [RAM84] Jonathan A. Rees, Norman I. Adams, and James R. Meehan. *The T Manual, Fourth Edition*. Yale University Computer Science Department, January 1984.
- [Rod92] Luis H. Rodriguez Jr. Towards a better understanding of compile-time mops for parallelizing compilers. In *Proceedings of the IMSA '92 Workshop on Reflection and Meta-level Architectures*, 1992. Also to appear in forthcoming PARC Technical Report.
- [SW80] Mary Shaw and Wm. A. Wulf. Towards relaxing assumptions in languages and their implementations. *SIGPLAN Notices*, 15(3):45–61, 1980.

- [WY88] Takuo Watanabe and Akinori Yonezawa. Reflection in an object-oriented concurrent language. In *Object Oriented Programming, Systems, Languages, and Applications Conference Proceedings*, pages 306–315, 1988.