What A Metaobject Protocol Based Compiler Can Do For Lisp

Gregor Kiczales John Lamping Anurag Mendhekar

January 31, 1994

Abstract

A new kind of metaobject protocol, that controls the compilation of programs, allows users to participate in the compilation in a principled and modular way. Such a compiler makes it possible to program in a high-level language and still maintain control over crucial implementation issues. This result is that a number of simple and elegant Scheme programs can be compiled as efficiently as if they had been written with special purpose primitives.

1 Introduction

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 and allows you to write your programs in a principled and modular way. But, on those crucial occasions where it is important that an abstraction be implemented in a particular way, you get to help the compiler achieve that end, also 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.

In this paper, we will show how a proof-of-concept metaobject protocol based compiler we have developed for Scheme allows programmers to cleanly, concisely and portably code the following examples, and have them work *as efficiently and as robustly as if they had been implemented primitively:*

- 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+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).
- CLOS-like method dispatch.
- 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.

Providing users with the kind of power these examples illustrate can significantly reduce the complexity of a number of programs. As an example, consider the implementation of CLOS in straight Common Lisp. While it is possible to implement a somewhat efficient CLOS in a small amount of code (10 pages or so) the fastest 'portable' implementation, PCL [KR90], is several hundred pages of code. Much of the bloat could be reduced if a compiler like the one we are developing were available.

Our compiler makes all this possible by using a metaobject protocol (MOP) to provide an auxiliary interface to the language that gives users the ability to incrementally modify the language's implementation and behavior. When working with our compiler, users focus the majority of their attention on writing a clean simple program in the base language (i.e. Scheme). They only occasionally turn to the metaobject protocol, to write meta-code that ensures their simple program compiles efficiently, or that extends the base language so their simple program can remain simple.

Previous metaobject protocols [KdRB91, BKK+86, Mae87, Coi87, WY88, WY91, Fer88, YTT89, MWY91, MMWY92, CM93] have been metacircular runtime implementations of an object-oriented language. The system presented here – as well as the related Anibus system[Rod91] – differs in several important ways. First, the base language our MOP controls, Scheme, is not object-oriented. As a result, the issues our MOP is concerned with are somewhat different; where earlier MOPs are concerned with the implementation of objects, our MOP is concerned with the implementation of primitive data types like pairs and procedures. Similarly, our MOP is concerned with ordinary procedure invocation, not method dispatch. Second, our MOP runs at compile-time, not run-time; the MOP controls the compilation of the Scheme program, rather than executing it directly. Finally, our MOP is not metacircular; the MOP is written in an object-oriented extension to Common Lisp, not in Scheme.

A principle goal in the design of our MOP is to give users the benefits of a high-level language like Scheme, together with the benefits of lower-level languages like C. Since the primary difference between these two languages is that Scheme provides higher-level data abstractions — both by being pointer based and by including pairs, integers, strings, procedures[1] and so on as primitive data types — our MOP design emphasizes user control of the layout of runtime data. As with any MOP, it also governs the implementation of Scheme control structures.

The focus of this paper is to illustrate the usefulness of a compiler based metaobject protocol. This paper includes only a sketch of the architecture of our MOP, we are currently preparing a companion paper that presents its detailed workings. This paper proceeds by successively presenting examples of Scheme programs that can be more efficient with our MOP-based compiler than with a traditional compiler. The structure of the MOP is presented in stages as we work through the examples.

[1] In keeping with the R^{4}RS terminology, we will use the term procedure throughout, even when the procedure is closed over free variables.

2 Procedures With Extra Data

As our first example, consider the case of a programmer who wants to associate an additional item of data with procedures. One such example is the generic functions of CLOS, which are first-class procedures with an associated list of methods and other information.

The details of the mechanism required depend somewhat on the use in question, but for purposes of exposition, assume that the user wants a new kind of `lambda` expression, called `Dlambda`, that returns procedures that can hold extra data. The extra data can be read and written using two new procedures, `procedure-data` and `procedure-set-data!`.

A natural way to implement this in Scheme is to use normal procedures as as the results of `Dlambda`s, and a global alist that maintains a correspondence from each procedure to its data.[2] The code that implements this is familiar. First, `Dlambda` would be defined as a macro, which would expand into something like:

```
(register-procedure-with-data (lambda (...) ...))
```

where the first argument is the lambda expression whose value will be returned and passed around. The procedures for accessing the data, together with `register-procedure-with-data` would be defined as follows:

```
(define procedure->data '())

(define register-procedure-with-data
  (lambda (procedure)
    (set! procedure->data
          (cons (cons procedure #f) procedure->data))
    procedure))

(define procedure-data
  (lambda (procedure)
    (cdr (assq procedure procedure->data))))

(define procedure-set-data!
  (lambda (procedure new)
    (set-cdr! (assq procedure procedure->data) new)))
```

That this code is so simple is a testament to the power of Lisp. But, this code has (at least) two major problems: First, the time required to lookup the data of a procedure is linear in the number of procedures. Second, the procedures and data cannot be garbage collected, even if they really are garbage to the rest of the program.

The efficiency problems could be addressed by rewriting the code to use hash-tables, except that vanilla Scheme doesn't supply them. To address the garbage collection problem as well, the hash

[2] An alternative implementation is also possible, in which the real procedure is wrapped in another one, that usually forwards calls to the real procedure, but can be called with special arguments to access the extra data. This strategy is better in some ways and worse in others than the one we will work with here. The point to note is that our MOP could be used to optimize that base Scheme program as well, in a manner similar to that shown in Section 7.

tables would have to be weak-pointered, but vanilla Scheme doesn't provide these either. But these suggestions skirt the obvious implementation, the one that would in all likelyhood have been chosen if procedures with extra data had been provided primitively: the data would be stored directly in the procedure object.

The question is, is it possible to allow the user to do this *without drowning them in the details of the compiler or the runtime.* To see how this can be done, consider that what the user actually wants to do is modify the *contract* that governs the implementation of procedures, to say that there should be an extra cell in each procedure, to hold the associated data. Consider that somewhere inside the compiler, whether explicit or implicit, there is inherently a declaration of the runtime format of procedures, which might look *something* like:

```
(procedure
  (nargs (int 8)) ... (code ptr) (environment ptr))
```

that is, a number of fields, some of which may be packed, some full pointers, some unique to a particular implementation, some an inherent part of all implementations and so on. Without having to understand the full structure definition, the user can reason about wanting to add one field, to get a structure definition something like:

```
(procedure
  (nargs (int 8)) ... (code ptr) (environment ptr) (data ptr))
```

to provide a place to store the data.

3 Decisions and The Granularity of Decision Making

While allowing users to change this structure definition might be easy enough, on closer examination, it isn't really what we want to do. The problem is that the scope of the change the user would be making is too large: all procedures would be affected, rather than only those created by a `Dlambda`.

But the scope of a change to procedure layout cannot just be the `lambda` that constructs the procedure. Both `lambda`s and applications must agree on the layout of procedures they have in common: where the code pointer is, how the environment is recorded, etc. In general, a `lambda` and an application must agree if the `lambda` might produce a procedure that will be invoked by the application.

Our MOP achieves this agreement by grouping applications and `lambda`s that must agree, and treating the group collectively. One contract governs the layout of all procedures manipulated by the group. User specialization of runtime data layout is thus specialization of these contracts. This contract arrangement is a natural consequence of a compile time MOP. Unlike a runtime MOP, where object dispatch can achieve agreement between the creator and the users of an object as uses occur, a compile time MOP must bring all concerned parties together at compile time to achieve agreement and avoid runtime dispatch.

Stepping back, a MOP gives programmers control over design and implementation issues that were previously out of their hands. But, to do this elegantly, a MOP designer must decide not only what previously hidden design decisions they want to give programmers control over, but also what

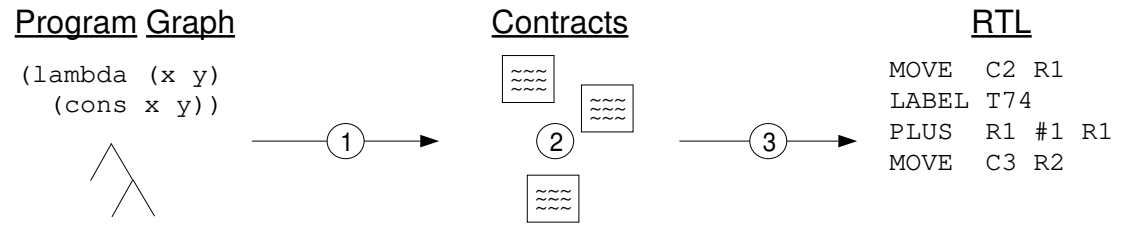

Figure 1: showing the overall architecture of the compiler.

the natural granularity of decision making is.[3] Getting this granularity right is what leads to good scope control. In our MOP, we want to expose decisions about layout of runtime data, and these groups are the natural grain size of the decision making. Contracts are the point where the decisions and the grain size come together.

4 The Architecture of the Metaobject Protocol

A metaobject protocol based compiler works by exposing some traditionally hidden analyses, data structures and decision making processes. But a primary goal of metaobject protocol design is to ensure that unimportant or idiosyncratic details remain hidden, and to expose only those issues that are both crucial and inherent to the compilation process — issues that any compiler will have to deal with.

A metaobject protocol works by using explicitly documented objects to represent the user's program, or information about the implementation of that program. These meta-level objects, or metaobjects, follow a well-documented protocol to compile (or in other systems execute) the user's program. The user can specialize the compiler by arranging for metaobjects that control appropriate parts of their program to have appropriately specialized methods that will be run at appropriate times during the course of compilation.

In our protocol there are three principle categories of metaobjects. *Program graph metaobjects* represent the source program, and are the inputs to the compiler. There is a single *contract metaobject* for every group of program graph metaobjects that must agree on the layout of runtime data. *RTL metaobjects* are the output of the MOP — they are statements in a simple register transfer language.

The work of the compiler, as controlled by our protocol, is divided into three phases, as shown in Figure 1. The first phase does flow analysis to determine what groups of program graph metaobjects must agree on runtime data they share, and creates a contract metaobject for each group. In the second phase each contract metaobject plans the layout it will use for the data it governs. The third phase generates the RTL, based on these decisions. There is a final translation of the RTL into primitive machine instructions, which is not under MOP control.

[3]It should be stressed that there is no such thing as a MOP that gives users access to *all decisions*. It simply isn't possible to do so. Metaobject protocols are intended to provide programmers with more flexibility, not infinite flexibility.

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.

4.1 Flow Analysis

The compiler accepts program graph metaobjects as inputs. There is a default class of program graph element corresponding to each kind of expression in Scheme, for example lambda expressions are instances of `<lambda-expr>`. Specialized methods can be attached to 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 this transfer of methods. See [Kic93] for details.)

The first phase of the compilation builds a contract metaobject for each group of program graph 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.

Contracts, in general, cover forms that construct and access (i.e. destruct) some kind 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 information used in grouping the program graph 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].

One issue that arises in the flow analysis is how to handle flows through code that the compiler does not have access to, code in other compilation units. Our prototype implementation is a block compiler; it skirts the issue by assuming that all code is available. One piece of future work is to explore extending existing module systems to include flow and contract information as part of modules so that it can be used in compiling other modules.

Since any flow analysis of Scheme is inherently imprecise, our default analyzer can lead to contracts that cover more program graph 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 Contract Planning

Once the contracts have been formed, a high-level protocol decides how the runtime data controlled by a contract will be laid out. Each program graph metaobject that participates in the contract can contribute methods to any of the steps of this protocol.

The contract first 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 kind of procedure contract might decide that it needs to store the code for the procedure, the procedure's environment, as well as several truly implementation-dependent values. Similarly, the default pair contract will probably decide that it must store two elements, in a two-word structure.

4.3 RTL Generation

For reasons of portability, our MOP generates machine-independent RTL that is later translated into native binary in a non-specializable way. The protocols that generate the RTL use destination driven code generation [DHB90] which enables tail-recursion optimizations and avoids the generation of many redundant moves and jumps.

RTL generation is done by a collaboration between the program graph and contract metaobjects. Since the contract metaobjects know the layout of runtime data structures (including environments), each contract is responsible for generating all code that creates or accesses the data structures that it covers. The program graph metaobjects, on the other hand, know the structure of the program, and are responsible for ensuring the connections between the individual data structure accesses.

The 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 program graph metaobjects for the procedure and for each of the arguments. The contract object requests each of those program graph 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 Procedures With Extra Data, Continued

Given this architecture, a user who wants to implement procedures with extra data must do two things: (i) Arrange for the contract of procedures returned by a `Dlambda` to have an extra cell for the data and (ii) arrange for `procedure-data` and `procedure-set-data!` to be able to access the extra cell. To do this, the user edits their base code as follows. First `Dlambdas` must be changed to expand to:

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

and the support procedures are edited as follows:

```
(define register-procedure-with-data
  (lambda (procedure)
```

```
    (set-procedure-data! procedure #f)))

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

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

Where the special annotation syntax with the curly braces indicates that the program graph metaobject that represents the immediately following expression has special methods named by the annotation. (Again, exactly how these methods are defined, associated with the annotation names, and propagated throughout the compilation is beyond the scope of this paper.)

There are four of these methods. The first is responsible for causing the generic placeholder reader and writer (`dummy-reader` and `dummy-writer`) to join the procedure contract as destructors of the procedure. The remaining methods all end up specializing the contract of the procedures in question. The first method runs during contract planning, it uses the symbolic layer of the protocol to request that an additional field be allocated in each procedure produced by the contract. This method looks roughly like:[4]

```
(method initialize (contract)                ;NOTE this
  (call-next-method)                         ;is
  (request-cell contract 'pointer 'slots))   ;metacode.
```

The second two methods also run at the symbolic layer of the protocol. They give the placeholders `dummy-reader` and `dummy-writer` their actual behavior. Specifically, they read and write the extra field that the first method requested be allocated. In all this metacode, the user isn't confronted with having to know exactly what layout the procedures has, or what else the rest of the compiler puts into them. They only ask for an extra pointer-sized field, and ask to be able to access it.

6 Immutable Structures

While Scheme doesn't provide a mechanism for ensuring that a user-defined structure will be truly opaque, such a thing can be quite useful, as a way to guarantee that other parts of the program won't make inappropriate access to a data abstraction. The degree of opacity desired can vary, including wanting to prohibit any access, or just wanting to prohibit side effects. Such immutability, itself, comes in various degrees. For example a user might like a structure to be mutable while it is being initialized, but then immutable afterwards, as in the following piece of code:

```
(define make-tandem
  (lambda ()
    {mutable-region}
    (let ((l {immutable-pair}(cons 'left #f))
          (r {immutable-pair}(cons 'right #f)))
      (set-cdr! l r)
```

[4] In our current implementation, these methods are not quite this clean, for a variety of uninteresting technical reasons. We are currently reworking the protocol details so that they will be this clean.

```
        (set-cdr! r l)
        l)))
  (define left cdr)
  (define right cdr)
```

This, admittedly somewhat silly, example defines `make-tandem` to return a "left-hand object," whose "right-hand object" can be fetched and vice versa. But, the opposite hands can't be changed once the objects have been created.

The {`immutable-pair`} annotation specializes the pair contract for each of the conses, with methods that override the normal code for processing `set-car!` and `set-cdr!`. The overriding methods simply check whether the call in question is within the scope of the `mutable-region` annotation, and if not signal an error at compile time.

Because the default flow analysis is imprecise, there can be cases where the flow analysis infers that a pair that was declared immutable might escape to some mutator, even though that can't actually happen. It would put the mutator in the same contract as the immutable pair, resulting in a compile time error compiling the mutator.

Thus, if the user sees such a compile time error, there are several possibilities. It may be that their program actually has this bug, and needs to be corrected. The user may know that the pair can't reach the mutator, in which case the flow analysis can be augmented with enough additional information to infer that. Finally, if the user isn't sure, they can change the behavior of {`immutable-pair`}s to emit code for mutators that does a run-time check.

This example could easily be enhanced, to ensure that no one outside the scope of the {`immutable-pair`} annotation was even allowed to call `car` or `cdr` on the pairs directly. (In which case the annotations would probably be renamed to be something like `opaque-pair`.)

This example points out that because our MOP's flow analyzer is extensible, it can be used to do powerful global analyses of programs, such as determining whether a pair might be side-effected, where datastructures can reach etc. In an environment like this it is crucial to give the user tools to understand these global interactions. Our prototype compiler includes a window based environment that gives point-and-click access to the metaobjects and their interconnections. This tool has proven to be invaluable in understanding the Scheme programs and metacode we are working with.

This is a good point to pause and contrast the MOP approach of letting the user add special features they require to the language, rather than the traditional approach of trying to include exactly the right features in the language in the first place. The problem with the later approach is that if every feature that anyone could want was in the language, the language would be too big. Further, a feature that is present might be more general that the user needs, leading to a less efficient implementation than necessary. Or it might not have quite the behavior the user wants, as illustrated by the range of conceivably useful behaviors for {`immutable-pair`}. The MOP approach lets the user craft the features they need, tuned to their requirements. Of course, in order for this approach to be viable, the new features need to run as fast as they would have if that had been provided by the language implementor. That's what the MOP is designed to allow.

7 Dispatching on Number of Parameters

A useful programming idiom is to be able to have a procedure that dispatches on the number of arguments it receives. This can be used in some ways of packaging readers and writers, and for

some functions that have optional arguments, such as:

```
(define increment
  (case-lambda
    ((n) (+ n 1))
    ((m n) (+ n m))))
```

This dispatching functionality is useful enough that some, but not all, Scheme implementations provide it. The way to implement `case-lambda` in vanilla Scheme is to define it as a macro, that expands something like the following:

```
(lambda .args.[5]
  (let ((.length. (length .args.)))
    (cond ((= .length. 1) (apply (lambda (n) (+ n 1)) .args.))
          ((= .length. 2) (apply (lambda (n m) (+ n m)) .args.))
          (otherwise (error ...)))))
```

Once again, that this code is so simple to write is a testament to the power of Scheme. But, that expressiveness comes at a cost. Unless the compiler includes a special mechanism for trying to detect this case, this code will run more slowly than one might like, first marshalling the arguments into a list, then taking the lenght of the list, and then dispatching on that length.

But for an application that is known to call a particular `case-lambda`, and that has a fixed number of arguments, the user would like the compiler to count the arguments at compile time and emit a call directly to the appropriate case. This is easy to express in our MOP, by relying on a documented optimization the compiler is already doing. During flow analysis, the compiler makes a list, for each application, of all the `lambda`s that might produce procedures that could be called from there. If there is only one `lambda` on the list, then emitted code calls the code for that `lambda` directly, rather than fetch the code pointer out of the procedure.

Given this, all that is needed to support `case-lambda` is to intercept the accumulation of possible `lambda`s and indicate that for a call to a `case-lambda` the lambda that is called is whichever inner one is selected by the case. The necessary method definition is:

```
(method destructor-flow (application case-lambda) ;NOTE this
  (if (fixed-arity? application)                  ;is
      (call-next-method                           ; metacode.
        application (which-inner-lambda case-lambda application))
      (call-next-method)))
```

This method runs as part of the flow analysis protocol, whenever the analyzer discovers that the result of a particular `case-lambda` can be called by a particular application. If this method can uniquely determine what case the application should call, it short circuits the analysis to that case (the first `call-next-method`). Otherwise, it allows the analysis to proceed normally. So, at the end of the analysis, every fixed-arity application that calls a particular `case-lambda` will be compiled as a direct jump to the appropriate case.

While this is not the most general optimization that can be done for `case-lambda`s, it does handle the most common case. Handling specific cases is exactly what the MOP is intended for.

[5] We aren't being hygenic here, not because we don't appreciate the value of cleanliness, but rather just to make the example easier to read – particularly for those who may not be able to read hygenic macros easily.

8 Exploiting the RTL

(Because of space limitations in this review version of the paper, this section has been substantially abridged.)

All of the examples presented so far have been expressible solely in terms of the higher level protocols for data structure layout and access. Those protocols themselves are built on top of the lower level protocols for emitting RTL. Some cases, such as providing interfaces to foreign code, require the the complete flexibility and low level specificity that comes with the ability to emit RTL directly. Another example is generating method dispatch code such as in PCL, where it is desirable to generate direct jumps and special bitwise operations.

There is crucial difference between the RTL protocol in our compiler and the ability to insert machine instructions that some languages and compilers support. Because the RTL protocol runs *as part of the compiler*, it has complete access to the compilation environment, and can exploit that information in generating the RTL. So, for example, it can know where a given variable value will be sitting, what the exact layout of a given datastructure will be and so on. Thus, the user can work with RTL either by writing high level code and controlling its compilation in RTL, or by directly generating RTL. In either case, their code will be much less brittle than a traditional machine code sequence sitting in the midst of higher level code.

9 Status and Future Work

The system we've described is operational, and runs all the examples presented in this paper. We have a window-oriented interface that is a great help in writing and debugging user meta-programs. The compiler itself is implemented in an object-oriented extension to Common Lisp called Traces [Kic93]. The RTL is translated to native SPARC code. To get our compiler up and running more quickly, we've chosen to try and get along without writing our own runtime, but instead working with an existing one. For the time being we are using the Lucid runtime, which is "compatible enough" with Scheme for our purposes. We simply link our binary code into the Lucid image and it runs fine.

But our current system is intended to be a proof of concept; it needs more engineering effort to be fully capable. First of all, it doesn't yet support all of Scheme; we have focused initially on facilities that presented an interesting mix of implementation choices. Also, our current translation from RTL to native binary weak; we don't generate high quality machine code.

We believe the right evaluation metric to use for this proof-of-concept MOP is a two step one: (i) Is the control our MOP provides useful, and, (ii) does it seem that we ought to be able to improve the quality of the runtime code generated. The examples in this paper demonstrate that the answer to the first metric is yes. As for the second, our approach is complementary to most of the work on code generation. The MOP focuses on front end issues and is compatible with the work that has been done on machine-independent RTL optimization and quality machine-dependent code generation. We expect that exploiting those techniques in our back end will give good quality code.

Three kinds of future work are needed. First, we need increase our library of examples, to improve our understanding of the style of working with such a system. This will help with our second goal, improving the design of the protocol so it is more powerful and easier to use. Our next key step in this direction is to solidify the symbolic layer of the protocol, so that more can be said at this level. Our third goal is to improve the quality of the final output code, primarily by improving

the code generator. We plan to explore whether we can connect to some existing Lisp backend, such as the CMU Python compiler, to take advantage of the implementation efforts of others.

10 Related Work

Metaobject protocols are a synthesis of procedural reflection and object-oriented techniques. Early work on procedural reflection focused on interpreters for imperative languages [Smi82, Smi84, dRS84, dR88]. The realization that object-oriented techniques provide a powerful mechanism for harnessing reflection led to explorations of user control over object representation, method dispatch, concurrency and distribution [Mae87, Coi87, WY88, WY91, Fer88]. Based on the intuition that opening a language's implementation to principled user intervention can improve performance, a number of researchers have pushed MOP techniques in this direction [BKK$^+$86, KdRB91, YTT89, Rod91, Rod92, MWY91, MMWY92]. [Kic92] presents a concise 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$^+$79]. An early summary of the motivations for providing this access can be found in [SW80].

11 Summary

While more work must be done before our compiler can be usable for day-to-day development, the work to date demonstrates that a metaobject protocol based compiler can give users useful control over the compilation of their programs, without drowning them in irrelevant details. It appears that this control can simplify existing programs. We estimate that functionality similar to that discussed in three examples of this paper — procedures with extra data, programmable flow analysis and direct generation of RTL for special code vectors — could reduce the amount of code in PCL by at least 20%. This makes us confident that we should proceed to the next stage of development for this compiler.

The traditional goal of language design has been to get the language — and its implementation — "just right," and then move on to build more functionality on top of it. But this is not always tractable. The reason is that it simply isn't possible to get an interface — or its implementation — just right for very long, or for very many different clients. We believe that this work presents a fundamental new approach approach in which the goal is to get the language to be "just adjustable"

In short, we've shown that adding metaobject protocol can move Lisp from being a "ball of mud that users can glom more mud onto" into a "piece of clay that can be scuplted to suit the needs of a far wider range of programmers than have traditionally used Lisp."

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+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.

[BKK+86] D.G. Bobrow, K. Kahn, G. Kiczales, L. Masinter, M. Stefik, and F. Zdybel. Commonloops: Merging Lisp and object-oriented programming. In *OOPSLA '86 Conference Proceedings, Sigplan Notices* **21***(11)*. ACM, Nov 1986.

[CM93] Shigeru Chiba and Takashi Masuda. Designing an extensible distributed language with a meta-level architecture. 1993.

[Coi87] Pierre Cointe. The ObjVlisp kernel: A reflexive lisp architecture to define a uniform object-oriented system. In Pattie Maes and Daniele Nardi, editors, *Meta-Level Architectures and Reflection*, pages 155–176. North-Holland, 1987.

[D+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.

[DHB90] R. K. Dybvig, R. Hieb, and T. Butler. Destination-driven code generation. Technical Report 302, Department of Computer Science, Indiana University, February 1990.

[dR88] Jim des Rivières. Control-related metalevel facilities in lisp. In Pattie Maes and Daniele Nardi, editors, *Meta-Level Architectures and Reflection*, pages 101–110. North-Holland, 1988.

[dRS84] Jim des Rivières and Brian Cantwell Smith. The implementation of procedurally reflective languages. In *Proceedings of the ACM Conference on LISP and Functional Programming*, pages 331–347. ACM, 1984.

[Dyb91] R. Kent Dybvig. *Chez Scheme System Manual, Revision 2.2, 1991*. Cadence Research Systems, Bloomington, Indiana, 1991.

[Fer88] Jacques Ferber. Conceptual reflection and actor language. In Pattie Maes and Daniele Nardi, editors, *Meta-Level Architectures and Reflection*, pages 177–193. North-Holland, 1988.

[Han91] Chris Hanson. Mit scheme reference manual. Technical Report Edition 1.1, Massachusetts Institute of Technology, November 1991. for Scheme Release 7.1.3.

[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.

[KR90] Gregor J. Kiczales and Luis H. Rodriguez Jr. Efficient method dispatch in PCL. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 99–105, 1990.

[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.

[Mae87] Pattie Maes. Concepts and experiments in computational reflection. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 147–155, 1987.

[MMWY92] Hidehiko Masuhara, Satoshi Matsuoka, Takuo Watanabe, and Akinori Yonezawa. Object-oriented concurrent reflective languages can be implemented efficiently. In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications*, pages 127–144, October 1992.

[MWY91] Satoshi Matsuoka, Takuo Watanabe, and Akinori Yonezawa. Hybrid group reflective architecture for object-oriented concurrent reflective programming. In *European Conference on Object Oriented Programming*, pages 231–250, 1991.

[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.

[Rod91] Luis H. Rodriguez Jr. Coarse-grained parallelism using metaobject protocols. Master's thesis, Massachusetts Institute of Technology, 1991.

[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.

[Smi82] Brian Cantwell Smith. Reflection and semantics in a procedural language (Ph. D. thesis). Technical Report TR-272, Laboratory for Computer Science, MIT, 1982.

[Smi84] Brian C. Smith. Reflection and semantics in LISP. In *Proceedings of the 11th Annual ACM Symposium on Principles of Programming Languages*, pages 23–35, 1984.

[SW80] Mary Shaw and Wm. A. Wulf. Towards relaxing assumptions in languages and their implementations. *SIGPLAN Notices*, 15(3):45–51, 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.

[WY91] Takuo Watanabe and Akinori Yonezawa. An actor-based metalevel architecture for group-wide reflection. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Proceedings of REX School/Workshop on Foundations of Object-Oriented Languages (REX/FOOL), Noordwijkerhout, the Netherlands, May, 1990*, number 489 in Lecture Notes in Computer Science, pages 405–425. Springer Verlag, 1991.

[YTT89] Yasuhiko Yokote, Fumio Teraoka, and Mario Tokoro. A reflective architecture for an object-oriented distributed operating system. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*, July 1989. (also available as a technical report SCSL-TR-89-001, Sony Computer Science Laboratory Inc.).