View-Based Abstraction
Enhancing Maintainability and Modularity in the Presence of Implementation Dependencies

by

Luis H. Rodriguez Jr.

S.M., S.B. C.S., Massachusetts Institute of Technology (1993)

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy in Computer Science
at the
Massachusetts Institute of Technology
September 1997

Copyright © Luis H. Rodriguez Jr., 1997.
All rights reserved.

Signature of Author _____

Department of Electrical Engineering and Computer Science
August 29, 1997

Certified by _____

Hal Abelson
Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by _____

Arthur C. Smith
Chairman, Departmental Committee on Graduate Students

View-Based Abstraction
Enhancing Maintainability and Modularity in the Presence of Implementation Dependencies

by

Luis H. Rodriguez Jr.

Submitted to the Department of Electrical Engineering and Computer Science on August 29, 1997 in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy in Computer Science

Abstract

This dissertation presents a new, backwards compatible, language independent, and incremental programming methodology called *view-based abstraction*. Unlike the well-known black-box abstraction approach, view-based abstraction enables programmers to maintain program modularity even in the presence of *implementation couplings*, i.e., dependencies among the code modules that rely on otherwise "hidden" implementation details not specified in the module interfaces. This dissertation also presents a transformation-based implementation of view-based abstraction, called *ViewForm*. ViewForm acts as a source-to-source preprocessor that automatically performs an implementation coupling expressed by the programmer. When the original code is later updated, ViewForm automatically attempts to reapply the implementation coupling to the updated code. ViewForm will modify the updated source code only if the coupling is still valid. In this way, by performing some extra work up front, the programmer performing an implementation coupling saves future programmers from having to pay for the consequences of broken modularity. To aid in writing this up-front ViewForm code, this dissertation presents a structured approach for using view-based abstraction and writing ViewForm transformations constructs.

To demonstrate view-based abstraction, ViewForm is used to produce automated, performance-based implementation couplings in three example programs: an amorphous computing simulator, a conditional-probability pedigree computation, and ViewForm itself. Unlike other approaches that also use interprocedural program analyses, the results indicate that view-based abstraction is practical and scales gracefully - the extra automation increased compilation time from a typical 34%, to 40% in the worst case, despite a less than fully optimized ViewForm implementation. Each optimization required the programmer to write only 65 to 137 lines of ViewForm code for programs of size 167 lines to 7,616 lines. This work is amortized as time saved by programmers modifying the original program in the future. In all three examples, ViewForm maintained modularity by regenerating correct code when the original modules were modified - even when those modifications were to the optimization-dependent sections of the original code.

Thesis Supervisor: Hal Abelson

Title: Professor of Computer Science and Engineering

Notice of Copyright and Terms of Limited License

This thesis document, including all figures, tables, and code fragments, is Copyright © Luis H. Rodriguez Jr., 1997. Country of first publication: United States of America. All rights granted to the author in accordance with 17 USC §§101 *et. seq.* are hereby reserved. This notice supersedes any other notice of copyright and terms of limited license for this document.

Pursuant to 17 USC §201(d)(2), the author hereby grants to the Massachusetts Institute of Technology (hereinafter "MIT" or "the Institute") certain non-exclusive, non-transferable, limited rights related to the copyright of this document:

1. MIT may reproduce paper copies of this thesis document for use within the MIT community for educational or research purposes (an action that is an exclusive right of the copyright holder under 17 USC §106(1)).
2. MIT may reproduce paper and microfiche copies of this thesis document for archival purposes within the MIT Library system (an action that is an exclusive right of the copyright holder under 17 USC §106(1), notwithstanding the provisions of 17 USC §108).
3. MIT may reproduce paper and electronic copies of the abstract page of this thesis (the page immediately preceding this page) and distribute such copies to the public (an action that is an exclusive right of the copyright holder under 17 USC §106(1) and 17 USC §106(3)), so long as no fee is charged for such copies.
4. MIT may reproduce paper copies of this thesis document and distribute such copies to the public (an action that is an exclusive right of the copyright holder under 17 USC §106(1) and 17 USC §106(3)) so long as no fee is charged for such copies in excess of the actual cost of making the copy.
5. All copies of this thesis document made by MIT under this license must include a copy of this license and the copyright notice on the title page.
6. All other uses of this thesis document within the scope of the exclusive rights of the copyright holder as specified in 17 USC §106 are reserved by the author, and any action by the Institute that infringes any of those exclusive rights, except as explicitly granted above, requires the expressed written consent of the author. In particular, MIT may not create an electronic version of this thesis document nor distribute an electronic version of this thesis document without the expressed written consent of the author.
7. Placement of this thesis within the collections of the MIT Library system constitutes acceptance of the terms of this license by MIT. MIT may cancel its agreement to the terms of this license by destroying all copies of this document made under the terms of this license and providing written notice to the author of this action.

Acknowledgments

A great number of people have contributed to the development of this work, both directly, and with equal importance, indirectly.

The simulator and pedigree examples in the dissertation are from Michael "Ziggy" Blair's thesis work. In addition, Blair's Scheme profiler was crucial for making ViewForm's performance practical. Stephen Adams provided hand-tailored x86-specific lap code that dramatically increased ViewForm's performance. Many thanks go to those who have built and supported MIT Scheme, especially the win32 port. Special thanks go to Alan Bawden, for a great discussion that led to insights on how Mini-ViewForm's views were intimately linked with abstraction.

Many thanks to Hal Abelson, Michael Blair, and John Guttag, for providing quick, honest, constructive, and detailed feedback on drafts of this dissertation. Thanks to Brian LaMacchia for the copyright notice on this dissertation.

Stephen Adams, Web Beebee, Michael Blair, Daniel Coore, Elmer Hung, Rajeev Surati, and Ron Weiss provided invaluable feedback for the thesis defense talk. Your support and feedback are deeply appreciated.

I am indebted to Gregor Kiczales, who took me under his wing while I was at Xerox PARC, and opened doors to places I might not otherwise have entered. Also at PARC, John Lamping's intellectual prowess gave me a high standard to which to strive. All the members of PARC's ECA group made my experiences there quite enjoyable, and contributed to where I am now.

Thanks to Todd Cass, for making the last five years of VI-A recruitment for PARC so enjoyable, and for exemplifying the kind of cool management style that is so easy to like.

Rajeev Surati has always been there. I will miss his recurring presence and his interesting and varied conversational topics. I will not miss the evil hack he played on me...

Daniel Coore was a great and genuine office mate who loved to enthusiastically discuss anything, anytime, anywhere. Thanks for the great discussions.

Many thanks to Ziggy, for diligently reading my writing throughout my graduate years at MIT, for feedback on my practice talks, and for freely providing me with code he spent countless months hacking. Ziggy also introduced me to high-quality beer. Yum!

Hal Abelson took me into MIT graduate school and Project MAC five years ago. That started me on the road to this dissertation. Thanks, Hal.

John Guttag set high standards and provided direct and honest feedback on my work and progress. The self-imposed "will Guttag like this?" litmus test helped me to better evaluate, focus, and direct my work. It was a pleasure having him on my thesis committee.

Special thanks to Dick Waters. His advice, perspective, and encouragement were crucial, instructive, and supportive. Correspondingly, thanks to Lyn Turback for recommending Dick Waters as a great person to have on my thesis committee.

Project MAC at MIT was an interesting place to do my research. The students and staff in the group leave with me a plethora of good memories, a few high-testosterone hobbies, and an invaluably thick shell. Thanks.

Certain MIT administrators (or those acting in an administrative context) made a significant impact on my stay at MIT. Among them are Tony Canchola-Flores and Eddie Grado. They demonstrated genuine interest

in me succeeding at MIT and *acted* on that interest. Instead of rhetoric, they reached out. Instead of excuses for why it could not be done, they produced results. Thank you for setting an example of what it should mean for students - all students - to receive "support".

My undergraduate academic advisor, Professor Joel Moses, pushed me to my academic limits early on; I'm thankful he acted, too. Lisa Bella in the Course Six office never ceased to provide a friendly smile and ear to a weary and tired 6.001 head lab TA, and thereafter.

Many people were instrumental and supportive in helping me work up the 6.001 ladder from student to recitation instructor. I enjoyed 6.001 at all these levels; it was some of the most fun I had at MIT. These people include Hal Abelson, Lou Braida, Arthur Gleckler, Eric Grimson, and Jim Miller.

Thanks to Mev, and the group at HP's MLL in Chelmsford for an interesting summer. Thanks also to Xuan Bui, Bob French, and Tor Ekqvist (from Hewlett Packard) for the industrial perspective and for discussing real examples of the implementation coupling problem in industry.

Kate and Editha - thanks for all the Melrose nights, the blading adventures, and for the reality checks. TYB! WWWWTG!

Yonald Chery and Chuck Rosenberg have been great confidants over the past ten years. I always feel better after seeing them. Thanks.

My late father and my mother taught me the joys of believing in myself instead of believing in image. By example, they taught me that giving to those you love and care for can be much more satisfying and fruitful than taking. This, their demanding work ethic, and their unconditional love and support have made them extraordinary role models, and the parents I will eternally be thankful for.

I do not know how to thank Christine Moore for all her help and support. She never hinted any complaints about the amount of time I devoted to this work. Instead, she supported me in every way (including plenty of outstanding gourmet meals!). She did what needed to be done, and then some. Rather than standing behind me, it was her shoulders that held me up high.

DARPA/NSF Acknowledgment

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for this research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-92-J-4097 and by the National Science Foundation under grant number MIP-9001651.

NSF Disclaimer

This material is based, in part, upon work supported under a National Science Foundation Graduate Fellowship. Any opinions, findings, conclusions or recommendations expressed in this publication are those of the author and do not necessarily reflect the views of the National Science Foundation.

Table of Contents

Abstract 3

Notice of Copyright and Terms of Limited License 4

Acknowledgments 5

Chapter 1 Introduction 15

1.1 Overview 15

1.2 Black Box Abstraction 17

1.3 View-Based Abstraction 21

1.4 ViewForm 23

1.5 Three Examples 24

1.6 Thesis Organization 26

Chapter 2 The Implementation Coupling Problem 27

2.1 The Dispatch Example 27

2.2 Black Box Abstraction 29

2.3 Desiderata 32

2.3.1 Backwards Compatibility 33

2.3.2 Incrementality 34

2.3.3 Language Independence 34

2.3.4 Ease of Understanding and Usability 34

2.3.5 Amortizable Time Savings 35

2.4 Influential Research Efforts 35

2.4.1 Reflection 35

2.4.2 Object Oriented Programming and MOPs 36

2.4.3 Program Slicing 37

2.4.4 Aspect-Oriented Programming 38

2.5 A Solution 38

Chapter 3 View-Based Abstraction 39

3.1 The Implementation Coupling Process 39

3.2 The Implementation-Coupling Steps 40

3.2.1 Step I - Boundary Identification 41

3.2.2 Step II - Precondition Development 41

3.2.3 Step III - Code Analysis 43

3.2.4 Step IV - Modification Development 44

3.2.5 Step V - Coupling Production 44

3.2.6 Step VI - Recoupling 45
3.3 Two Critical Ideas 45
3.4 View-Based Abstraction 46
3.5 Summary of the View-Based Abstraction Model 47
3.6 Summary of View-Based Abstraction Methodology 47
3.7 View-Based Abstraction Components 48
3.7.1 Views 48
3.7.2 Contexts 50
3.7.3 Coupling Language 50
3.7.3.1 Predicates 50
3.7.3.2 Actions 51
3.7.3.3 Dispatchers 51
3.7.3.4 Rules 52
3.7.3.5 Couplers 53
3.7.3.6 View Invalidation/Recoupler 53
3.8 View-Based Abstraction Methodology 54
3.8.1 Step i - Context Identification 54
3.8.2 Step ii - Predicate Development 54
3.8.3 Step iii - View Development 54
3.8.4 Step iv - Action Development 55
3.8.5 Step v - Coupler Development 55
3.8.6 Step vi - Invalidation/Recoupling 55

Chapter 4 ViewForm 57

4.1 ViewForm Overview Scenario 57
4.2 Viewcode 59
4.3 Contexts 60
4.4 ViewForm Coupling Constructs 62
4.4.1 Characteristic Vform Interface 62
4.4.2 Predicates 62
4.4.3 Actions 64
4.4.4 Dispatchers 64
4.4.5 Rules and Couplers 65
4.4.5.1 Couplers 66
4.5 Vforms 67
4.6 Views 69
4.6.1 Viewcode View 71

4.6.2 Alpha View 72
4.6.3 Liar View 73
4.6.4 Higher-Order Views 75
4.7 Explanations 77
4.8 View Invalidation/Recoupler 78
4.9 Dispatcher Example 79
4.9.1 Boundary/Context Identification 79
4.9.2 Precondition/Predicate Development 79
4.9.3 Code Analysis/View Development 81
4.9.4 Modification/Action Development 82
4.9.5 Coupling/Coupler Production 83
4.9.6 Invalidation/Recoupling 84
4.10 User Interaction 85
4.11 Complexity Layering 85

Chapter 5 Examples and Analysis 87

5.1 Testing Methodology 87
5.1.1 Testing Process and Evaluation Metrics 88
5.2 Amorphous Computing Simulation 91
5.2.1 Amorphous Computing Simulator Background 91
5.2.2 Building the Coupling 92
5.2.3 Preconditions and Predicates 93
5.2.4 Actions 97
5.2.5 Dispatcher and Coupler 99
5.2.6 Invoking the View Invalidation/Recoupler 101
5.2.7 Discussion 102
5.2.8 Quantitative Measurements 103
5.3 Pedigree Example 104
5.3.1 Short-Circuiting Multiplication 105
5.3.2 Function Inlining 107
5.3.3 Implementing the Short-Circuiting Multiplication Rule 107
5.3.3.1 Modules/Contexts 107
5.3.3.2 Preconditions/Predicates 108
5.3.3.3 Views 108
5.3.3.4 Modifications/Actions 108
5.3.3.5 A Dispatcher 109
5.3.3.6 A Rule 109

5.3.4 Comb View 110
5.3.5 Function Inlining 112
5.3.5.1 Modules/Contexts 113
5.3.5.2 Preconditions/Predicates 113
5.3.5.3 A Modification and Action 115
5.3.5.4 A Dispatcher 115
5.3.5.5 A Rule 115
5.3.6 Combining the Rules into a Coupler 116
5.3.7 ViewForm Generated Code 117
5.3.8 Modifying the Pedigree Code 118
5.3.9 Quantitative Measurements 118
5.4 ViewForm Example 119
5.4.1 Modules/Contexts 122
5.4.2 Preconditions/Predicates 122
5.4.3 Views 123
5.4.4 Modifications/Actions 124
5.4.5 Dispatchers and Couplers 125
5.4.6 Platform View 127
5.4.7 ViewForm Output 128
5.4.8 Quantitative Measurements 130
5.5 Analysis and Evaluation 131
5.5.1 The Desired Couplings Metric 131
5.5.1.1 The Simulator Example 131
5.5.1.2 The Pedigree Example 132
5.5.1.3 The ViewForm Example 133
5.5.2 The Desiderata Metric 133
5.5.2.1 Backwards Compatibility 133
5.5.2.2 Incrementality 133
5.5.2.3 Language Independence 134
5.5.2.4 Ease of Understanding and Usability 135
5.5.2.5 Amortizable Time Savings 136
5.6 Lessons Learned 138
5.6.1 Determining Preconditions and Program Modifications 138
5.6.2 Conservative Program Analyses 138
5.6.3 Default Program Analyses 139
5.6.4 Default Coupling Library 139

5.6.5 View Selection and Computation 139
5.7 View Updates 140
5.7.1 View Consistency During Rule Invocations 140
5.8 Source Code Maintenance 141
5.8.1 Interface Non-Preserving 141
5.8.2 Interface Preserving 142
5.9 User Interaction 142
5.9.0.1 User Interaction Reduces Fragility 144
5.10 Source Code Availability 144
5.10.1 Source Code Somewhat Available but Partially Hidden 144
5.10.2 Not Available and Not Hidden 145
5.11 Fundamental Insights 145

Chapter 6 Related and Future Work 147

6.1 Views 147
6.1.1 The View Oriented Model 147
6.1.2 PECAN 148
6.1.3 Semantic Program Graphs 149
6.1.4 Documenting Programs Through Views 150
6.2 Special-Purpose Languages via Transformations 150
6.2.1 TXL 150
6.2.2 ASCENT 151
6.2.3 Proxac 151
6.2.4 Elaborations 151
6.2.5 Darlington's User-Interactive Transformation System 151
6.2.6 PECOS and LIBRA 152
6.2.7 Cheatham's Transformation System for Reuse 152
6.2.8 CIP 152
6.3 Open Implementations 153
6.3.1 Intrigue 153
6.3.2 Anibus 154
6.3.3 Data Path Macros 154
6.4 Optimizing Programs via Transformations 155
6.4.1 Dora 155
6.4.2 GENesis 155
6.4.3 Program Optimization and Derivation via Transformations 156
6.5 Interactive Program Design and Construction 156

6.5.1 KBEmacs 156
6.5.2 A Program Verifier Assistant 156
6.5.3 CCEL 157
6.6 Future Work 157
6.6.1 Reusability 157
6.6.2 Symmetry in Couplings 158
6.6.3 Other Programming Paradigms 158
6.6.4 An Experiment 159

Chapter 7 Conclusion 161

7.1 Summary 161
7.2 Contributions 162
7.3 Conclusion 164

Appendix A Selected Aspects of the ViewForm Interface 165

A.1 ViewForm Expression Type Testers 165
A.2 Viewcode Expression Return Types 165
A.3 Miscellaneous ViewForm Functions 166
A.4 Viewcode Canonicalization 166

Appendix B Implementation of Selected ViewForm Functions 167

Bibliography 173

Table of Figures

Figure 1-1 - Black-Box Abstraction....17
Figure 1-2 - Uncoupled dispatch-expression....18
Figure 1-3 - Coupling Implementation....18
Figure 1-4 - Desired Coupled Implementation....19
Figure 1-5 - Invalidating Implementation....19
Figure 1-6 - Black-Box Abstraction Implementation Coupling....20
Figure 1-7 - View-Based Abstraction....20
Figure 1-8 - Actual Coupled Code Generated by ViewForm....23
Figure 2-1 - Formalized Implementation Coupling....32
Figure 3-1 - Implementation-Coupling Steps....40
Figure 3-2 - Partial Data Flow Analysis on f....49
Figure 3-3 - Conditional Rewrite Rule....52
Figure 4-1 - ViewForm....58
Figure 5-1 - Particle Data Abstraction Implementation, adapted by Blair from code by Abelson....90
Figure 5-2 - Optimized Particle Value Implementation....91
Figure 5-3 - ViewForm Output....92
Figure 5-4 - Simulator Example Predicates....93
Figure 5-5 - Simulator Example Actions....98
Figure 5-6 - Simulator Example Dispatchers and Other Rules....100
Figure 5-7 - Pedigree Computation Code....104
Figure 5-8 - Pedigree Data Abstraction Implementation....105
Figure 5-9 - Desired P_pedigree Implementation....106

Chapter 1

Introduction

This dissertation is for programmers on software development teams who want to improve performance by using implementation dependencies that break modularity. View-based abstraction is a new abstraction model that otherwise preserves modularity in the presence of implementation dependencies. Unlike meta-level or transformation-based approaches, view-based abstraction is practical, is backwards compatible with black-box abstraction, is independent of the source language and its implementation, and supports non-local program analysis, non-semantics-preserving modifications, and user interaction.

1.1 Overview

Black-box abstraction is a well-known and principled mechanism for designing, maintaining and implementing software.[20] Under black-box abstraction, an application is reasoned about as a combination of *modules* (i.e., black boxes). Each module hides its implementation under an *interface*. A module's implementation can depend only on another module's interface, and cannot depend on another module's implementation. This simple rule gives rise to *modularity*, the ability to replace any module with any other module that implements an identical interface. Modularity, in turn, helps reduce the number of dependencies within an application, thereby enhancing software maintainability. It does not take much programming experience to realize, however, that violating this rule can lead to gains in performance[52,69,33,59,53] and extensibility[90]. Such gains are realized by modifying module implementations to depend directly upon otherwise hidden implementation details. This results in what I term *implementation coupling*; a dependency from one module's implementation to another's. Unfortunately, implementation coupling is a kind of abstraction violation and breaks modularity, thereby resulting in an application that is substantially more difficult to maintain, port, extend, and debug in the future. The programmer who performs such an implementation coupling (and breaks modularity) is therefore not necessarily the programmer who must pay for the consequences of broken modularity.

The programmers who must pay for consequences of broken modularity are future programmers who would have been able to otherwise update and modify the code in a modular way.

This dissertation's goal is to address this problem of enhancing modularity and maintainability in the presence of implementation couplings, in a practical setting. The approach used to solve the problem is to have the programmer who produces an implementation coupling perform some extra work. This extra work will provide effective modularity for future maintenance programmers, and will thus save them from having to deduce the existence of any implementation couplings, the modules the couplings affect, the way the couplings interact with the code, and a way of patching or reverting the couplings if necessary.

To solve the implementation coupling problem using this approach, the implementation-coupling problem is first formalized. Then, a six-step model of the implementation-coupling process is developed. This dissertation then demonstrates how the implementation-coupling problem can be reduced to the problem of automating the six steps (more specifically, automating the final step). This reduction is the crux of the dissertation, and leads to view-based abstraction. Under view-based abstraction, a programmer automates an implementation coupling by expressing the coupling in terms of a set of coupling constructs. Each construct corresponds to a step in the six-step model. These constructs are concretely provided by ViewForm, a transformation-based implementation of view-based abstraction. Once an implementation coupling is expressed, ViewForm can automatically generate the coupled code from the original source code (the results of which can then be passed immediately to the compiler). Of more importance, when the original source code is later updated, ViewForm will attempt to regenerate the implementation coupling on the updated source code. ViewForm will regenerate the coupling only if it is still valid with respect to the updated source code. This regeneration step effectively preserves modularity by hiding (i.e., abstracting) implementation couplings from future programmers.

As discussed in Chapter 2, view-based abstraction provides backwards compatibility, incrementality, language independence, ease of use, and amortizable time savings. To demonstrate view-based abstraction and these properties, ViewForm is used to express performance-based implementation couplings (i.e., modularity-breaking optimizations) on three example programs: an amorphous computation simulator, a conditional-probability pedigree computation, and ViewForm itself. The results show how view-based abstraction permits implementation coupling without precluding modularity, while attaining the five properties listed above. The results also indicate that view-based abstraction, unlike other approaches, is practical in three respects. The first is that the extra automation increased the overall compilation time from a typical 34%, up to 40% in the worst case. This worst-case time was despite the use of a full interprocedural data-flow

analysis and a less than fully optimized ViewForm implementation. A second indication of practicality is that each optimization required only 65 to 137 lines of ViewForm code, in the form of 8 to 13 "rules". These rules included code templates describing each respective optimization. The ability to scale gracefully is a third concern for practicality. The results demonstrate that scaling issues can be addressed in three ways. First, by limiting interprocedural analysis to only relevant modules, the amount of code that must be analyzed is reduced. Second, by building views with varying computational resource needs, a ViewForm rule can make the most of the available resources. Third, user interaction can provide views with assumptions not decidable or otherwise computable given a practical amount of resources. Even in the worst case, ViewForm will still generate correct code, though not necessarily coupled code (in which case ViewForm can automatically alert the programmer). This worst case is far better than what can happen under black-box abstraction. The worst case for implementation coupling under black-box abstraction is incorrect code and no automatic notification to the programmer.

In all three examples, ViewForm was able to maintain modularity by regenerating correct code when the original, dependent modules were modified, even when those modifications were to the optimization-dependent sections of the original code. The remainder of this chapter expands on this summary of the implementation-coupling problem, the solution, the implementation, and the results.

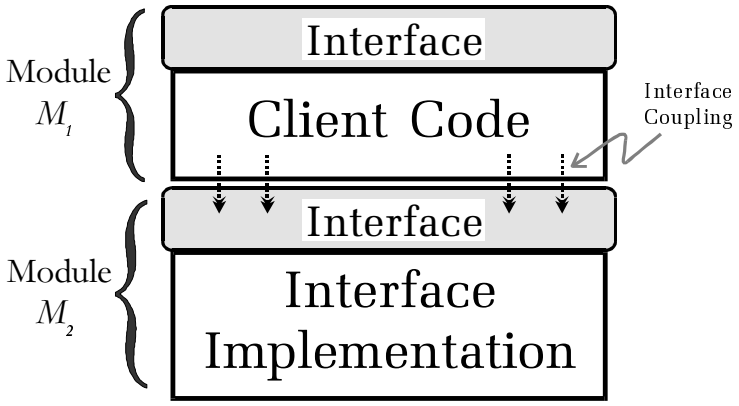

Figure 1-1 - Black-Box Abstraction

1.2 Black Box Abstraction

Figure 1-1 illustrates a typical use of black-box abstraction. M_2's implementation is hidden under its interface, which is simply a collection of names and specifications. M_1's implementation depends upon this interface, meaning M_1 is *interface coupled* to M_2 (i.e., M_1 refers to at least one

```
(define dispatch-expression
  (lambda (exp)
    (cond ((type? 'literal       exp) (process-literal       exp))
          ((type? 'java-name     exp) (process-java-name     exp))
          ((type? 'new           exp) (process-new           exp))
          ((type? 'dot           exp) (process-dot           exp))
          ((type? 'call          exp) (process-call          exp))
          ((type? 'cast          exp) (process-cast          exp))
          ((type? 'instanceof    exp) (process-instanceof    exp))
          ((type? 'built-in-expr exp) (process-built-in-expr exp))
          ((type? 'assignment    exp) (process-assignment    exp))
          (else (error "Unknown Type: " exp)))))

(define (process-literal       exp) ...)
(define (process-java-name     exp) ...)
(define (process-new           exp) ...)
(define (process-dot           exp) ...)
(define (process-call          exp) ...)
(define (process-cast          exp) ...)
(define (process-instanceof    exp) ...)
(define (process-built-in-expr exp) ...)
(define (process-assignment    exp) ...)
```

Figure 1-2 - Uncoupled `dispatch-expression`

```
;;;
;;; type? is an interface function
;;;

(define-integrable type?
  (lambda (type-keyword exp)
    (eq? type-keyword (exp-type exp))))

;;;
;;; Nothing below this line is exported
;;;

(define (exp-type exp)
  (let ((result (assq exp *exp-type*)))
    (if result
        (cdr result)
        result)))

(define *exp-type* ...)
```

Figure 1-3 - Coupling Implementation

of the names specified in M_2's interface). M_1 does not, however, depend on M_2's implementation. Black-box abstraction provides modularity precisely because M_1's implementation does not depend on M_2's hidden implementation details. Accordingly, M_2's implementation can be modified without affecting M_1's interface, so long as M_2's interface remains valid. Modularity makes it easier for current and future programmers to maintain, extend, port, and debug code.

Modularity does not, however, always make it easier to optimize code. This can be exemplified by the code in Figure 1-2, a linear-time dispatch on decaf-java expressions[65] (this code has been simplified to illustrate the problem; substantial examples are presented later in Chapter 5). This dispatch code, which is in a module corresponding to M_1 above, calls the `type?` function. `type?` is in a different module whose code is partially given Figure 1-3. `type?`'s module corresponds to M_2 above. As a result of modularity, a programmer can change `type?`'s implementation without even knowing about `dispatch-expression`, and can change `dispatch-expression` without having to know how `type?` is implemented. Now, suppose we want to optimize `dispatch-expression`. In par-

```
(define dispatch-expression
   (let ((dispatch-table (make-symbol-hash-table)))
     (for-each (lambda (symbol proc)
                 (hash-table/put! dispatch-table symbol proc))
               (list 'literal  'java-name  'new  'dot     'call
                     'cast    'instanceof 'built-in-expr 'assignment)
               (list process-literal    process-java-name process-new
                     process-dot        process-call      process-cast
                     process-instanceof process-built-in-expr
                     process-assignment))
     (lambda (exp)
       (let ((proc (hash-table/get dispatch-table (exp-type exp) #f)))
         (if proc
             (proc exp)
             (error "Unknown Type: " exp))))))
```

Figure 1-4 - Desired Coupled Implementation

ticular, instead of a linear-time dispatch, we want an expected constant-time dispatch. This can be accomplished using hash tables, as shown in Figure 1-4. Therein, however, lies a problem.

While `dispatch-expression` in Figure 1-4 meets the optimization criteria we set out, it does so at the expense of modularity. Instead of calling `type?`, `dispatch-expression` now calls `exp-type`, an otherwise hidden part of `type?`'s implementation. Since Scheme[49] has no module interface checking (like a variety of other languages), the reference to `exp-type` goes unchecked.* The result is an additional dependency that consequently increases the program's complexity. This dependency means that `dispatch-expression`'s module (M_d) is now *implementation coupled* to `type?`'s module (M_t). Even so, `dispatch-expression`'s interface is still valid, and is now faster. The problem introduced by the implementation coupling will not rear itself until either M_d or M_t is subsequently modified.

To demonstrate how the implementation coupling precludes modularity, let us suppose that a maintenance programmer is now told to optimize `type?`. This programmer would likely notice that `exp-type` uses a linear search. This search can be optimized by using an expected constant-time search instead. The code in Figure 1-5 demonstrates an implementation of this criterion using hash tables. The problem with the code in Figure 1-5 is that it invalidates the optimized `dispatch-expression` in Figure 1-4. Since the optimized `dispatch-expression` uses `exp-type` (which no longer exists), `dispatch-expression` will not be

```
(define-integrable (type? type-keyword expression)
  (eq? type-keyword (fast-exp-type expression)))

(define (fast-exp-type exp)
  (hash-table/get *exp-type* exp #f))

(define *exp-type* ...)
```

Figure 1-5 - Invalidating Implementation

* In languages that do have module interface checking, `exp-type` would have to be added to the module's interface, even though it is not part of the original module's interface specification. Nevertheless, requiring module interface checking is neither backwards compatible nor necessary to solve the problem.

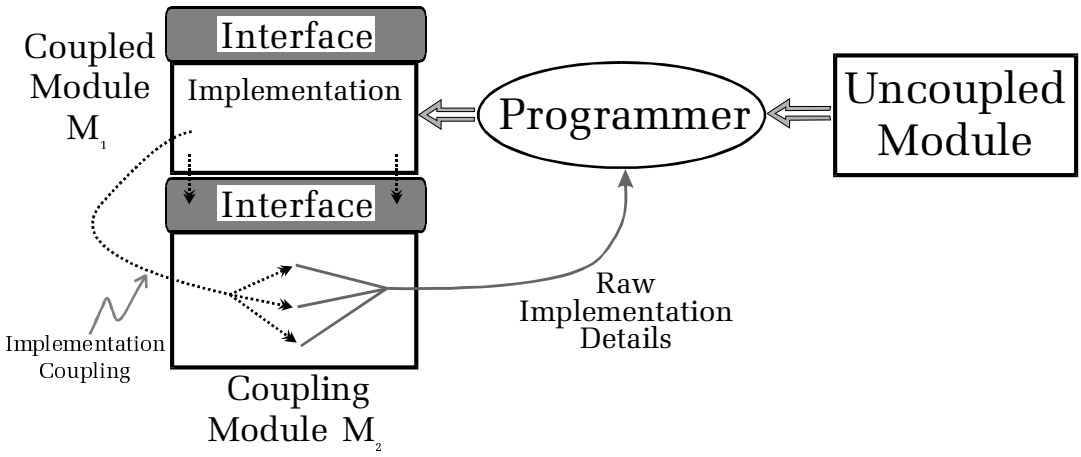

Figure 1-6 - Black-Box Abstraction Implementation Coupling

able to predictably determine an expression's type. The maintenance programmer working solely on making `type?` faster will be disappointed to learn that somewhere in the application, one or more pieces of code in different modules depend on `exp-type` (a function not in M_t's interface). With modularity precluded, the programmer must either find and deal with these dependencies, or try looking for some other way to optimize M_t.

While simple, the dispatch example demonstrates the essence of the implementation-coupling problem. Instead of corresponding to M_1 and M_2 in Figure 1-1, M_d and M_t respectively correspond to M_1 and M_2 in Figure 1-6. This figure illustrates a programmer using knowledge of M_t's implementation (i.e., the function `exp-type`) to build the optimized `dispatch-expression` from the unoptimized version. The result is an implementation-coupled version of `dispatch-expression` that can easily be invalidated by a future modification, such as that in Figure 1-5.

A solution to the implementation-coupling problem will permit implementation coupling without precluding modularity. For the dispatch example, this means that when `type?` is changed

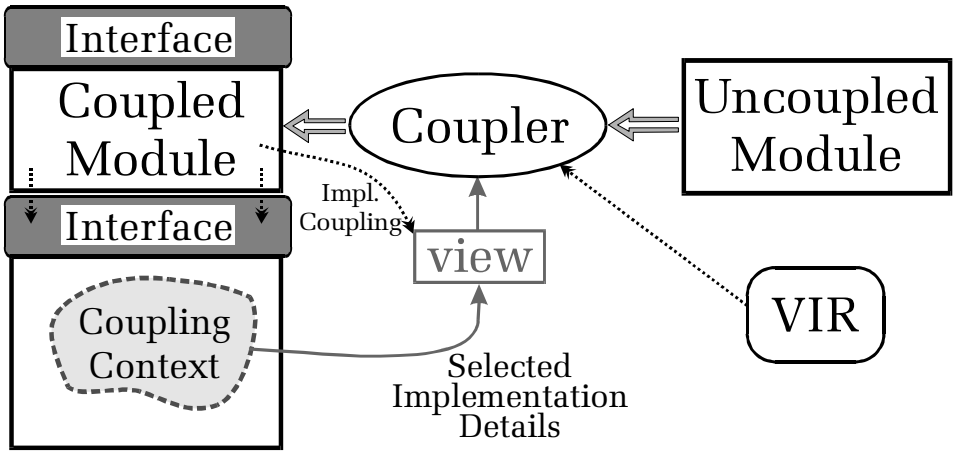

Figure 1-7 - View-Based Abstraction

to use the optimized `fast-exp-type`, the solution must notice the change and determine that the change invalidates the implementation-coupled `dispatch-expression`. The invalid `dispatch-expression` must then be replaced with either the original version or with a version that uses `fast-exp-type`. The latter outcome is more desirable than the former, as is an automatic solution that performs these steps when `type?` is modified.

Chapter 2 describes this problem in more detail, presents a formalization of the problem, and discusses influential related work.

1.3 View-Based Abstraction

This dissertation proposes view-based abstraction as a solution to the implementation-coupling problem. View-based abstraction is based on a model I developed that describes implementation coupling as a six-step process. View-based abstraction provides a methodology for imperatively expressing three of these steps. By doing so, the other three steps can be automated. It is this automation that enables view-based abstraction to provide modularity in the presence of implementation coupling. These six coupling steps, given below, are explained in greater detail in Chapter 3.

I. *Boundary Identification* - Identify the abstraction boundaries in the code

II. *Precondition Development* - Develop the preconditions under which the desired coupling is valid

III. *Code Analysis* - Analyze the code, to gather the program properties needed to validate the preconditions

IV. *Modification Development* - Develop code modifications for changing the uncoupled code into the desired code

V. *Coupling Production* - If the preconditions are valid, carry out the code modifications to produce coupled code

VI. *Recoupling* - Whenever the coupling code or the uncoupled code is modified in the future, redo the coupling steps to produce a new, valid piece of coupled code

The first five steps correspond to the process by which the dispatch example was implementation coupled (see Figure 1-6).

From this process-based model of implementation coupling, I developed two critical ideas that led to view-based abstraction. The first is given below:

- Modularity is precluded by an implementation coupling only if that coupling is allowed to persist through future code modifications that render the coupling invalid

For instance, in the dispatch example, modularity is precluded only if `dispatch-expression`'s use of `exp-type` is allowed to persist after M_t is modified to use `fast-exp-type`. Black-box abstraction does not require this persistence check, most likely because it is impractical to require a programmer to perform the necessary examinations each time a module is modified. From this idea, one can conclude that being able to *automatically* carry out the recoupling step would provide the guarantee needed to maintain modularity. This is because the recoupling step ensures that a coupling persists through a future code modification only if the coupling is valid in the context of that modification. A maintenance programmer would thus be free to modularly modify a program in the presence of implementation couplings, leaving the work of ensuring every coupling's subsequent validity to the computer.

The real power of this six-step implementation-coupling model and the first idea drawn from it is that together, they allow the implementation-coupling problem to be reduced to the problem of automating the recoupling step. This is now a tangible problem, and leads to the second critical idea underlying view-based abstraction:

- The recoupling step can be automated if the code analysis and coupling production steps are automated, and these steps can be automated if the boundary identification, precondition development, and modification development steps are expressed in an imperative programming language.

Providing this automation and a language for expressing implementation couplings leads to view-based abstraction.

View-based abstraction consists of a model and a methodology mirroring the six-step implementation process. The model introduces various components: *contexts* (for representing abstraction boundaries), *predicates* (for expressing preconditions), *actions* (for expressing program modifications), *dispatchers* (correspond to rewrite rules), *rules* (combinations of these constructs), *couplers* (for automatically generating a coupling), and *views* (mappings between program expressions and program properties). The model also requires a *view invalidation/recoupler*, responsible for automatically carrying out the recoupling step. The view-based abstraction methodology is a process that is layered over the six-step implementation-coupling process. By mirroring the six-step process, the methodology provides a structured approach to using the view-based abstraction model (including its constructs) to produce implementation couplings.

View-based abstraction is illustrated in Figure 1-7. The programmer is replaced by a coupler, and the recoupling step is performed by the view invalidation/recoupler. For the dispatch example, the implementation coupling from M_d to M_t would be registered with the view invalidation/recoupler. If M_t was subsequently modified to use `fast-exp-type`, the view invalida-

tion/recoupler would recognize this change, take the original implementation of `dispatch-expression`, and produce a new version that used `fast-exp-type` instead of `exp-type`.

The above is a summary of the view-based abstraction model and methodology, which are presented in detail in Chapter 3.

```
(define dispatch-expression
  (let ((exp-type-symbol (lambda (exp) (exp-type exp)))
        (dispatch-table (make-symbol-hash-table)))
   (for-each (lambda (symbol process-exp)
               (hash-table/put! dispatch-table symbol process-exp))
             (list (quote literal)    (quote java-name)
                   (quote new)        (quote dot)
                   (quote call)       (quote cast)
                   (quote instanceof) (quote built-in-expr)
                   (quote assignment))
             (list process-literal    process-java-name
                   process-new        process-dot
                   process-call       process-cast
                   process-instanceof process-built-in-expr
                   process-assignment))
   (lambda (exp)
    (let ((process-exp (hash-table/get dispatch-table (exp-type-symbol exp) ())))
      (if process-exp
          (process-exp exp)
          (begin (error "Unknown Type: " exp)))))))
```

Figure 1-8 - Actual Coupled Code Generated by ViewForm

1.4 ViewForm

ViewForm is my implementation of view-based abstraction, and was developed to experiment with the view-based abstraction model and methodology. ViewForm is an imperative, transformation-based language layered over Scheme[49]. ViewForm performs source-to-source transformations on Scheme code augmented with various MIT Scheme[42] constructs. ViewForm also introduces a novel construct, the *vform*. Vforms are combinable, delegation-based constructs that operate on program expressions, with respect to a context. Vforms are the basis for constructing predicates, actions, dispatchers, rules, couplers, and views.

In addition to the view-based abstraction coupling constructs, ViewForm implements various default views, including a data-flow view and variable naming view. ViewForm also implements a simple but fully functioning view invalidation/recoupler. Within ViewForm, *complexity layering* makes more common types of implementation couplings easier to implement. ViewForm maintains backwards compatibility with current software engineering practice, provides incrementality, and demonstrates view-based abstraction's viability. Chapter 4 presents View-

Form in more detail, including a complete ViewForm implementation for solving the dispatch example.

1.5 Three Examples

The code in Figure 1-8 is what ViewForm generated for the dispatch example. The dispatch example ViewForm code, presented in Chapter 4, exemplifies the view-based abstraction methodology as well as techniques used later on the three example programs. The examples are: an amorphous computation simulator from [7], a conditional-probability pedigree computation program [83], and ViewForm itself. These three example programs show how view-based abstraction can be used to solve instances of the implementation-coupling problem in a more practical setting.

Each of the three examples examines different issues within the view-based abstraction framework. The simulator example tests how ViewForm performs on source code written by someone other than myself, applying an implementation coupling developed by someone other than myself. The pedigree example (whose source code was also written by someone other than myself), illustrates what happens when a desired view is not provided by default. The ViewForm example brings up the issues of scale and user interaction, since ViewForm is considerably larger than either of the first two examples (~7600 lines of code). While most practical programs are much larger than ViewForm, this example, and the previous ones, are used to demonstrate that the size of a program is not indicative of the amount of code that must actually be analyzed to apply an implementation coupling. The ViewForm example also demonstrates how user interaction can be used in conjunction with view-based abstraction, and how this affects the kinds and scales of views that are feasible to perform.

The results show how view-based abstraction permits implementation coupling without precluding modularity while attaining backwards compatibility, language independence, incrementality, ease of use, and amortizable time savings. The most important property of these five is backwards compatibility. This is because the problem would be much simpler if we could produce a solution that requires changes to the source language's specification or implementation (e.g., requiring native module support), or to the source code (e.g., annotations with module information). This dissertation demonstrates that requiring changes to the language, the language implementation, or the source code is not necessary.

The results also indicate that view-based abstraction is practical, for three reasons. The first is that the extra automation for the examples increased the overall compilation time from a typical 34%, to 40% in the worst case. This worst-case overhead was despite the use of a full, interprocedural, LIAR-style[79] data-flow analysis view and a less than fully optimized ViewForm im-

plementation. This worst-case overhead was measured on the ViewForm example, running the data-flow analysis over the entire program. This was not actually necessary, as only 5 of ViewForm's 28 modules needed to be analyzed for the desired coupling. The decision to analyze all 28 modules was made to project ViewForm's performance on much larger programs, where an analysis might be needed on 7600 lines out of tens or hundreds of thousands of lines of code. In this hypothetical larger case, the 40% overhead would also be reduced.

A second indication of ViewForm's practicality is that each optimization required only 65 to 137 lines of ViewForm code, in the form of 8 to 13 ViewForm "rules". These rules included code templates describing each respective optimization. Some rules were implemented from scratch using ViewForm primitive functions, others were implemented using ViewForm library functions, while others turned out general enough to become part of the ViewForm library. In all cases, the process of implementing the rules (i.e., the view-based abstraction methodology) corresponded to the six-step process outlined above. This process provides a structured and methodical approach that simplifies rule implementation.

The ability to scale gracefully is a third concern for practicality. The results demonstrate that scaling issues can be addressed in three ways. First, by limiting the scope of the interprocedural analysis to only relevant modules, the amount of code that must be analyzed is reduced. For example, in the simulator example, only 1,100 out of over 10,000 lines of code needed to be analyzed. Likewise, in the ViewForm example, only 5 out of the 28 files needed to be analyzed (although, as discussed above, all 28 were analyzed to test scaling issues). This kind of scope reduction should be possible in many cases, since typical human programmers do not (and for practical reasons, cannot) precisely analyze huge amounts of code when manually producing an implementation coupling.

Second, with respect to scaling, by utilizing views with varying computational resource requirements, a ViewForm rule can make the most of the available resources. The ViewForm example demonstrates how the data-flow analysis view accomplished this by making space and time tradeoffs to maximize computational resource utilization.

Third, with respect to scale, user interaction can provide views with assumptions not decidable or otherwise computable given a practical amount of resources. Even in the worst case scaling scenario, however, ViewForm will still generate correct code, though not necessarily coupled code (this newly generated uncoupled code will not necessarily be slower, and may even be faster than the previously coupled code). In addition, if ViewForm generates uncoupled code, ViewForm can automatically alert the programmer to this fact. This worst case is far better than what can happen under black-box abstraction. The worst case for implementation coupling under black-box abstraction is incorrect code and no automatic notification for the programmer.

In all three examples, ViewForm was able to maintain modularity by regenerating correct code when the original, dependent modules were modified, even when those modifications were to the optimization-dependent sections of the original code.

1.6 Thesis Organization

The remainder of the thesis discusses and elaborates on the ideas and issues introduced above. Chapter 2 describes and formalizes the implementation-coupling problem, and reviews influential and foundational research efforts. Chapter 3 then presents view-based abstraction in depth. Chapter 4 follows with the ViewForm transformation language. In Chapter 5, ViewForm is used to couple three example programs, after which the results are qualified, quantified, discussed, and analyzed. Other related research efforts and ideas for future work are presented in Chapter 6. Chapter 7 concludes the dissertation, providing a summary of the work and elaborating this dissertation's contributions.

Chapter 2

The Implementation Coupling Problem

The implementation coupling problem is that implementation couplings break modularity. To solve this problem, modularity must be preserved in the presence of implementation couplings. This chapter discusses the problem in more detail, relating it back to the dispatch example presented in Chapter 1. The discussion includes a formalization of the problem and of various terms that will be used throughout this dissertation. A set of desiderata for the solution is also given, and other research efforts most closely related to the problem are examined.

2.1 The Dispatch Example

The code in Figure 1-2 and Figure 1-3 typifies a scenario faced by programmers using black-box abstraction. The code in Figure 1-2 is similar to what may be found in an interpreter, compiler, specializer, program analyzer, or any of a variety of applications taking source-level input. For this particular code, `dispatch-expression` is based on "decaf java-in-Scheme," found in [65]. `dispatch-expression` takes a decaf-java expression, determines its type using `type?` and a candidate type symbol, then processes the expression based on its determined type. This process is also known as a *type dispatch*. A decaf-java expression's type is computed using the function `type?`, a sample implementation of which is given in Figure 1-3. `type?` takes a decaf-java expression and a type symbol, and returns a true value if the type symbol represents the expression's type.

In this scenario, the application has been designed with `dispatch-expression` and `type?` in separate modules (i.e., black boxes). This is to allow their respective implementations to be modified in a modular way. Modularity, in this context, means the ability to replace any implementation with any other that correctly implements the module's interface.

Given these implementations of `dispatch-expression` and `type?`, suppose that a programmer is given the task of optimizing `dispatch-expression`. The current implementation dispatches in O(n) time, where n is the number of decaf-java expression types (more of which could be added

later). A more efficient scheme, based on [58], is to use a hash table to attain O(1) expected time. This new implementation computes an expression's type symbol, maps this symbol to a `process-<exp>` procedure using a hash table, then invokes the `process-<exp>` procedure on the expression. An implementation of this version of `dispatch-expression` is given in Figure 1-4. The problem with this code is that it violates `type?`'s module boundary by depending on `type?`'s implementation. In particular, the new `dispatch-expression` code calls `exp-type`, a function called by `type?` to compute an expression's type symbol. This is an abstraction violation, and is not allowed by black-box abstraction. The real problem, however, has not yet manifested itself in an observable way. In fact, up to now, `dispatch-expression`'s implementation remains faithful to its interface, despite the obvious implementation dependency. This illustrates how easily implementation dependencies can make their way into an application: they can be innocuous and unobservable (except when they speed up a program, in which case they are considered a feature and not a bug!).

There may come a time, however, when the costs of these abstraction violation dependencies become apparent. In the dispatch example, this cost manifests itself when a future programmer, believing that modularity has not been precluded, tries increasing `type?`'s performance. In the original code, `type?` calls `exp-type`, which uses an association list to look up an expression's type. This is an O(n) process in the number of expressions. The code in Figure 1-5, in contrast, uses `fast-exp-type` to provide the type lookup for `type?`. `fast-exp-type` uses a hash table to look up an expression's type, reducing the lookup to an expected O(1) time. Once this new, faster implementation of `type?` is installed, `exp-type` will no longer exist.[†] This means that the optimized `dispatch-expression` (in Figure 1-4) now has a bug: it can no longer determine an expression's type, and therefore will fail or will return unpredictable results. Thus, the optimized `dispatch-expression`'s dependency on `type?`'s implementation has broken modularity. The cost of this broken modularity must now be paid by this future programmer and other, subsequent programmers updating the dispatch code. These programmers must deduce the existing implementation dependencies, determine which other modules are affected, determine how the dependencies interact with any desired code updates, and determine how to revert or patch the code to make the code updates consistent with the dependencies.

This scenario illustrates exactly the kind of problem this dissertation addresses. The goal of this dissertation is to show how maintainability and modularity can be preserved in the presence of these kinds of implementation dependencies. This problem is formalized below, as part of a discussion of black-box abstraction.

[†] Although lisp-like languages will not necessarily eliminate `exp-type`'s name and value, the association list it uses for type lookup will no longer be updated. The result is the same kind of problem - a bug.

2.2 Black Box Abstraction

Figure 1-1 illustrates a typical use of black-box abstraction. Module M_2 is hiding its implementation under an interface, and Module M_1's implementation depends on that interface. Module M_1 also has its own interface, which other modules may depend upon. Black-box abstraction provides modularity precisely because M_1's implementation does not depend on M_2's hidden implementation details. This means that M_2's implementation can be modified without affecting M_1's interface or implementation, so long as M_2's interface remains valid. In order to establish a more precise foundation for these ideas and terms, their meanings are formalized in a language-independent way as follows:

$P = \{M^*\}$

A program is composed of a set of modules

$M = < Int, Impl >$

Modules consist of an interface and implementation

$Impl(M) = \{exp^*\}$

An *implementation* is a set of language expressions whose syntax and meaning depend on the language's syntax and semantics

$Int(M) = \{spec^*\}$

An *interface* is a set of specifications

$spec = < ident, desc >$

Specifications consist of an identifier and a description

$ident = (identifier)$

An *identifier* is a language-dependent name, used to reference a piece of functionality

$desc = (description)$

A *description* is a language-dependent description of the functionality referenced by an identifier

With respect to the dispatch example, the program contains at least two modules: M_d, whose implementation is the code in Figure 1-2 and M_t, whose implementation is the code in Figure 1-3. M_d's interface has one specification whose identifier is `dispatch-expression`. Its description says that `dispatch-expression` is invoked on a decaf-java code representation and returns the result of interpreting that code representation. M_t's interface also has one specification. Its identifier is `type?`, and its description says that `type?` is invoked on two arguments, a type symbol and a decaf-java code expression. `type?` returns true if the decaf-java expression is of the type represented by the type symbol.

Given these equation-based definitions for programs, modules, interfaces, implementations, and specifications, the notion of dependencies can be formalized. This dissertation is concerned with two kinds of dependencies, referred to as *interface couplings* and *implementation couplings*. The former is discussed first.

$Impl(M_1) \mapsto Int(M_2) \equiv$

$\exists\ ident_i \in \{ident(spec_1)\ \ldots\ ident(spec_n)\}$

such that $Int(M_2) = \{spec_1 \ldots spec_n\}$ and $\{ident_i\} \subseteq Impl(M_1)$

This says that if M_1's implementation refers to an identifier in M_2's interface, $M_1$is said to be *interface coupled* to M_2

$Dep(M) \equiv \{M_1, \cdots, M_n\}$

such that $\forall_{i=1,\cdots,n} Impl(M) \mapsto Int(M_i)$

A module M depends on a module M_i if M is interface coupled to M_i. The set of all such modules M_i is $Dep(M)$

$ValidImpl(Int, Impl, \{M_1 \ldots M_n\}) \equiv$

$\forall_{i=1-n}, ValidModule(M_i)$ and

Impl is a valid implementation of *Int* with respect to the interfaces $Int(M_1) \ldots Int(M_n)$

$ValidImpl(Int, Impl, \varnothing) \equiv$ *Impl* is a valid implementation of *Int*

An implementation can be valid only if it genuinely implements its interface and if its dependent modules (if any) are valid

$ValidModule(M) \equiv ValidImpl(Int(M), Impl(M), Dep(M))$

A valid module is one with a valid implementation

$ValidProg(P) \equiv \forall_{M \in P} ValidModule(M)$

A program is valid if its modules are valid

The first of the five equations above defines the meaning of an interface coupling. For the dispatch example, the interface coupling from M_d to M_t happens because `dispatch-expression` in M_d's implementation refers to the identifier `type?` in M_t's interface specification. The remaining equations above specify what it means for an implementation, a module, or a program to be valid. It is worthy noting that the fundamental notion of "validity" - as in an implementation being valid with respect to its interface - is specific to a language's semantics. A language-dependent version of the equations above would also provide language-specific definitions for program expressions, identifiers, specifications, and possibly a VALID valuation function (i.e., [16] provides these for Scheme). Nevertheless, the formalization above is at the level of detail necessary to discuss the problem addressed by this dissertation.

The more important form of coupling for this dissertation is implementation coupling. The definition below defines an implementation coupling, additionally providing the definition of an invalidating implementation with respect to two modules.

$InvalidatingImpl(M_1, M_2) \equiv \{Impl'^*\}$ such that

$M_2 \in Dep(M_1)$ and

$ValidImpl(Int(M_2), Impl', Dep(M_2))$ and

$\neg ValidImpl(Int(M_1), Impl(M_1), \{< Int(M_2), Impl' >\} \bigcup (Dep(M_1) - M_2))$

$Impl(M_1) \mapsto Impl(M_2) \equiv$

$\neg InvalidatingImpl(M_1, M_2) = \emptyset$

M_1 is *implementation coupled* to M_2 if there exists a valid implementation of M_2's interface which invalidates M_1.

The intuition behind these definitions is based upon the notion of modularity. The first equation defines an invalidating implementation as one that, under black-box abstraction, is a valid implementation of an interface (i.e., M_2's interface) but whose presence will nevertheless invalidate a different module (i.e., M_1). This means that modularity, or the ability to replace M_2's implementation by any other valid implementation of its interface, is precluded. The second equation defines the existence of such an invalidating implementation between two interface-coupled modules as an *implementation coupling*. For instance, in the dispatch example, the *coupled module* whose implementation is given in Figure 1-4, $M_{d \to t}$, is implementation coupled to the *coupling module* M_t (i.e., $Impl(M_{d \to t}) \mapsto Impl(M_t)$) because there exists an invalidating implementation of M_t with respect to $M_{d \to t}$. One possible invalidating implementation, *Impl'*, in this set $InvalidatingImpl(M_{d \to t}, M_t)$ is given in Figure 1-5. *Impl'* is a valid implementation of M_t's interface (i.e., $ValidImpl(Int(M_t), Impl', Dep(M_t))$). If used, however, *Impl'* would invalidate $M_{d \to t}$'s interface since the definition of `dispatch-expression` in $M_{d \to t}$ calls `exp-type` (from M_t) instead of `fast-exp-type` (from *Impl'*).

Given these definitions, the implementation-coupling problem can now be specified. Suppose we have four modules, M_1, M_2, M_1', M_2', and a program, P,with the properties and relationships given in Figure 2-1 (these equations are true for the dispatch example if P is the implementation-coupled program containing the code in Figure 1-4 and Figure 1-3, M_1 is M_d, M_2 is M_t, $Impl_1$ is the code in Figure 1-4 and $Impl_2$ is the code in Figure 1-5). These equations specify an interface coupling from M_1 to M_2, and an implementation coupling from M_1' to M_2. This happens, for example, when a programmer takes the implementation of M_1 and modifies it into M_1', an implementation that depends on M_2's hidden implementation details. M_2' is a valid replacement of M_2 that is perhaps the result of routine program maintenance on M_2's implementation. If M_2' were to replace M_2, however, M_1' would become an invalid module, since M_2' is an invalidating implementation of the implementation coupling from M_1' to M_2.

$M_1, M_2, M_1', M_2' =$ Modules
$Impl_1, Impl_2 = Implementations$
$P = \{M_1', M_2\}$
$M_1' =< Int(M_1), Impl_1 >$
$M_2' =< Int(M_2), Impl_2 >$
$ValidModule(M_1)$
$ValidModule(M_2)$
$Impl(M_1) \neq Impl_1$
$ValidImpl(Int(M_1), Impl_1, Dep(M_1))$
$ValidImpl(Int(M_2), Impl_2, Dep(M_2))$
$Int(M_1) \mapsto Impl(M_2)$
$Impl_1 \mapsto Impl(M_2)$
$Impl_2 \in InvalidatingImpl(M_1', M_2)$

Figure 2-1 - Formalized Implementation Coupling

The *implementation-coupling problem* is that modularity is precluded in the presence of an implementation coupling, as formalized above and demonstrated in the dispatch example. This means that in the presence of an implementation coupling, a future programmer is not free to replace or modify a module's implementation even if the result is an otherwise valid implementation of the module's interface. Instead, a future programmer must first meticulously examine otherwise hidden (and possibly complex) implementation details to find any existing implementation couplings. The programmer must then ensure that these couplings are consistent with the changes he or she wants to make. If the changes are not consistent, the programmer must either remove the implementation couplings (and recursively act on those changes) or not make any changes at all. This deters maintainability, making an application more difficult to debug, port, optimize, or extend.

A solution to this problem should nevertheless permit implementations to be coupled, in order to gain the associated performance benefits. The solution must also preserve modularity by allowing M_2 to be replaced with M_2' in P, in a way that does not invalidate P. In the case of the dispatch example, this means being able to freely replace the implementation of `type?` in Figure 1-3 with the code in Figure 1-5, even if `dispatch-expression` is implemented as in Figure 1-4, all without invalidating P.

More formally, a solution must satisfy the equation, $P[M_2'/M_2] \Rightarrow ValidProg(P)$, with respect to the equations in Figure 2-1. This equation states that any code modifications allowed under black-box abstraction, when in the presence of implementation coupling, should not invalidate a program. To truly solve the problem, maintenance programmers should not have to reason about existing couplings, and should not have to know if any even exist.

2.3 Desiderata

A trivial "solution" to the implementation-coupling problem is to program without modules. At the opposite extreme is an ideal, but as of yet, unattainable solution: automating the art of producing optimized programs. Given that the former solution is where programming began and the latter is where programming may someday go, one can imagine a variety of ways of addressing the implementation-coupling problem along this spectrum. Given this variety, it is im-

portant to outline a set of metrics by which to evaluate this dissertation's solution as well as other, existing approaches to the implementation-coupling problem. While these desiderata make the problem more difficult to solve, their purpose is to ensure a conforming solution can yield insight into the problem's true subtleties, pragmatic issues, and tradeoffs.

Overall, the main goal of a good solution should be to help push the state of the art in the direction of the ideal solution mentioned above. Towards this goal, this dissertation lists five properties that a solution to the implementation-coupling problem should have. These properties are backwards compatibility, incrementality, language independence, ease of understanding and usability, and amortizable time savings. None of the work surveyed in this dissertation combines these properties into one solution, and few provide any form of backwards compatibility.

2.3.1 Backwards Compatibility

Backwards compatibility means the implementation-coupling solution should be consistent with black-box abstraction and should not depend on a particular feature of a current programming paradigm. In particular, backwards compatibility prohibits all of the following. While the first three items could be classified as something done only once, the last two items cannot and are more fundamental.

- Modifications to the source language
- Modifications to any source language implementation
- Rewriting the application into a different language(s)
- Modifications to the uncoupled program (including pragmas or annotations)
- Requiring full, direct access to all of the uncoupled application's source code

If backwards compatibility were not required, a solution would be easier to come by. For example, one way to solve the implementation-coupling problem is to design an otherwise contrived (and restrictive) source language in which couplings are easy to automatically detect. While this approach is valid from the standpoint of learning more about the problem, the approach leaves some of the more interesting, practical, and difficult aspects of the implementation-coupling problem unaddressed. The solution presented in this dissertation illustrates this point.

Another approach that is not backwards compatible requires users to decorate their code with machine-readable annotations or specifications. This usually complicates code maintenance (i.e., "What do I do with the code decorations if I modify the code?", "How does my change affect the specification?"), and is not necessarily compatible with existing programming environment components, such as source-code editors and program analyzers.

2.3.2 Incrementality

For this dissertation, incrementality means that a given implementation-coupling solution is not pervasive. That is, in order for the solution to work, it need not be applied to every piece of source code or every implementation coupling in an application. This allows programmers to selectively apply the solution to code and couplings where they personally judge the benefits to outweigh any costs. Incrementality also allows the solution to be resilient in cases when an application harbors implementation couplings unbeknown to the programmer.

Without incrementality, one possible solution to the implementation coupling problem is to pervasively decorate every application interface with machine-readable specifications. These specifications would then be used to validate couplings whenever any part of the implementation was modified. This would assume that the application contained no couplings to begin with. Besides not being backwards compatible, this kind of approach would not necessarily be resilient in the presence of existing or unknown couplings. In addition, this approach would require a disproportional amount of work in the case when the coupling modifications were small compared to the overall size of the application (which must be decorated).

2.3.3 Language Independence

Language independence means the solution does not depend on the presence of a source-language property. Without language independence, a solution will not necessarily be more generally applicable. The solution, for instance, should not depend on the source language being object oriented, functional, or strongly typed. While these kinds of language properties can make it easier to find or express implementation couplings, making them as part of the solution is an artificial requirement that is not necessary.

Language independence does not mean the solution is equally powerful in all languages. A solution might provide better results in a more constrained or less expressive language than in a very unconstrained or expressive language.

2.3.4 Ease of Understanding and Usability

The solution should provide an abstraction model that is easy to understand, even if it means pushing complexity into the model's implementation. In addition, programmers should not have to be language experts to understand how the model works, although language experts should not be limited to using the model in ways that inexperienced programmers use it. Ease of understanding and usability also mean not having to learn or design new languages from scratch for each coupling. This includes not requiring programmers to change their coding styles to accommodate the solution.

2.3.5 Amortizable Time Savings

Any extra time spent applying the solution to a coupling should be recoverable as saved time when an application undergoes repeated future maintenance, modifications, and updates. The extra, up-front time spent applying the solution can then be amortized over each future change to the application's code. This requirement precludes any solution that might require a programmer to manually find existing couplings, or manually validate the persistence of any existing couplings. This requirement implies a solution that is mostly or completely automated.

2.4 Influential Research Efforts

Various aspects of previous research efforts address the implementation coupling problem directly or indirectly. The efforts that most influenced this dissertation are described below.

2.4.1 Reflection

Programming languages can provide reflective support via additional constructs in a base language.[82] These constructs semantically shift the language's subject matter, usually to that of the language implementation. This means that using these constructs, a programmer can write code that reasons about its own execution. For example, some constructs can *reify* the program's environment and stack, making them available as modifiable data structures. This approach of using reflection to control an implementation has made its way into various programming paradigms, such as distributed programming[68], real-time programming[44], and many others[96]. In each of these cases, the desire to control an application's implementation was otherwise precluded by the lack of language support. Reflective constructs provide this support, thus enabling programmers to express knowledge of the underlying application's implementation as part of the source code. While a language's implementation is more constrained in the presence of these constructs, one goal of reflective language design is to reduce the impact of these constraints.

Using reflection to solve the dispatch example would first require a Scheme implementation that had been extended to support reflective constructs. The code for `dispatch-expression` would then need to be modified in this reflective Scheme so that it had access to `type?`'s implementation. This implies some way of acquiring a representation of `type?`'s implementation, after which the reflective code could substantially modify `dispatch-expression`'s behavior. As this example demonstrates, there are two significant problems with using reflection on its own to solve the implementation coupling problem in a practical way. The first is the need to add reflective constructs to a language specification. The second is the need to modify every language implementation upon which the code will run to support these new constructs. These needs are inconsis-

tent with backwards compatibility. Moreover, the surveyed literature did not contain any examples of reflection being used to directly solve instances of the implementation-coupling problem.

2.4.2 Object Oriented Programming and MOPs

CLOS[85] is an object oriented language that supports multiple inheritance. A class that multiply inherits from a class structure, for example, is coupled to all of the inherited methods that are combined to form class-specific generic function effective methods. Thus, the computed effective methods are implementation coupled to the methods found throughout the class's superclasses. Modularity is maintained through an explicit invalidation mechanism. When a superclass's implementation is modified, all affected effective methods are invalidated and recomputed (usually lazily, for better overall performance). [58] contains a description of a method memoization mechanism which automatically and efficiently ensures that effective methods are recomputed when their implementation-coupled dependencies are invalidated.

A solution to the implementation coupling problem should likewise maintain modularity by providing a mechanism for automatically managing any couplings. The mechanism should automatically invalidate and recompute the implementation couplings when necessary. In the dispatch example, for instance, replacing M_t with an invalidating implementation should cause $M_{d\to t}$ to become invalid and should subsequently cause a valid version of `dispatch-expression` to be generated. In order to keep the solution general, however, the mechanism should not be linked to a particular language or programming paradigm.

A metaobject protocol, or MOP, opens an application's implementation by allowing a user to control important aspects of the implementation's decision-making process.[55] This control is provided by documenting a set of metaobjects and a protocol that the implementation adheres to. A metaobject is an object that determines some aspect of an implementation's behavior, usually by making a critical decision. The protocol describes the interaction between the metaobjects themselves and between the metaobjects and other parts of the implementation. MOPs are designed to provide the user with both locality and incrementality, among other benefits.

MOPs provide two interfaces for each module: a primary interface and a secondary interface. The former is the usual functionality-based interface, while the latter allows the module's implementation to be altered. Most MOP research is aimed at increasing performance, although some has been aimed at making a language's semantics extensible[90].

Intrigue[59] is a MOP for controlling the implementation of a Scheme compiler. The Intrigue MOP provides access to the compiler's data-flow analysis engine. This engine is very flexibly programmable to allow the propagation of user-defined quantities interprocedurally throughout the compiler's internal representation. This engine was used to implement datapath macros[56].

Datapath macros are transformation constructs that operate on a flow graph program representation as opposed to operating on textually-localized program text.

Intrigue and datapath macros can go a long way towards solving the dispatch example. A datapath macro could analyze the `dispatch-expression` and `type?` code, after which the macro could modify `dispatch-expression` into the desired, coupled code. While this approach can produce the desired modification, it is not enough to solve the implementation-coupling problem. One reason is that it does not guarantee modularity in the presence of an implementation coupling. That is, if `type?`'s implementation is changed, the optimized `dispatch-expression`'s implementation could be made invalid. Another reason is that datapath macros do not address the issue of scale. In larger examples, a general flow analysis may be impractical over the entire program.

2.4.3 Program Slicing

A program slice[94] with respect to a construct, c, includes all constructs that might affect the variable values used by c. Program slicing can be used in a variety of situations, such as program manipulation and representation frameworks[73], the automatic recovery of reusable components[60], merging pieces of code derived from a common, base piece of code [46], and determining whether a code modification is consistent with the original code[34]. Slicing allows programmers to abstract away details unrelated to the construct they are currently interested in. It also allows programmers to consider statements that are not just textually localized in a file. These properties are essential to a solution for the implementation-coupling problem.

The work in [46] provides an interesting perspective on the implementation-coupling problem. This work develops a mechanism for combining two programs derived from a common, base program. If the two derived programs are consistent and combinable, the given program slicing techniques will automatically produce a merged program. If the programs cannot be merged, no combined program is produced.

A related piece of work, [34], approaches this merging problem from a different perspective. Instead of combining two semantically unconstrained programs derived from a common base, the approach is to constrain future modifications to the original derived program so that the modifications are all consistent with the derived program.

Nevertheless, neither of these approaches solves the implementation coupling problem. In both cases, the result is either a successful merging or no merging at all. A solution to the implementation coupling problem, however, requires that a future programmer's modifications always work. That is, they must take precedence over any existing implementation couplings in case of a conflict (so as to preserve modularity). A second problem is practicality. Since neither

work presents examples or time measurements, it is difficult to gauge the amount of computational resources required to perform a merging.

2.4.4 Aspect-Oriented Programming

Aspect-oriented programming[57] is a relatively new abstraction model for conceptually separating programming concerns[48] from each other. Some sample concerns identified in [48] include: class organization, synchronization, location control, real-time constraints, and failure recovery. An aspect-oriented program consists of distinct pieces of code, each addressing a different concern. The base concern program is written in a mainstream language or a variant thereof, and expresses the application's essential functionality. The other concerns are written in appropriate aspect languages. An Aspect Weaver™ then combines the concerns into code that can be interpreted, compiled, or otherwise executed. One goal of aspect-oriented programming is to decouple conceptually distinct concerns from the base concern, so that each concern can be reasoned about as independent of the base concern as possible.

Aspect-oriented programming does not address the implementation-coupling problem directly because it does not provide a way to validate implementation dependencies (including those induced by the aspect programs) when the main program or aspect programs are modified. Aspect-oriented programming can, however, push those dependencies to the aspect layer. In the dispatch example, for instance, the algorithm used to map a decaf-java expression type to a `process-<exp>` expression could be specified by an aspect. Similarly, the way that `type?` computes a type symbol could also be specified by a different aspect. The former aspect could depend on the latter, and the aspect weaver could then be run to ensure that any future modifications get propagated out to the target code that is actually run. Unless the aspect language provides some sort of aspect modularity, however, the implementation-coupling problem will remain, albeit at a different level. One other drawback to using aspect-oriented programming for the implementation-coupling problem is that aspect languages must currently be custom-designed or at least specialized for each application. This may not continue to be the case as more research into aspect-oriented programming is carried out.

2.5 A Solution

This dissertation solves the implementation-coupling problem using the view-based abstraction model and methodology, presented next in Chapter 3. View-based abstraction attains the desiderata described in this chapter, while maintaining modularity in the presence of implementation couplings.

Chapter 3

View-Based Abstraction

This dissertation proposes view-based abstraction as a solution to the implementation-coupling problem presented in Chapter 2. View-based abstraction is based on a model I developed which describes implementation coupling as a six-step process. View-based abstraction provides a methodology for imperatively expressing these steps so that they can be automated. It is this automation that enables view-based abstraction to provide modularity in the presence of implementation coupling.

This chapter presents these six steps, describes the view-based abstraction model and methodology, builds upon the formal definitions presented in Chapter 2, and relates it all back to the dispatch example.

3.1 The Implementation Coupling Process

The insights that led to view-based abstraction were derived from an examination of the implementation-coupling process itself. This examination led to an implementation-coupling model that breaks down the implementation-coupling process into a set of six distinct (though not necessarily sequential) steps. This "divide and conquer" approach is novel with respect to the surveyed literature, and it leads to the two critical ideas upon which this dissertation's thesis is based. While many valid implementation-coupling models are possible, the one presented here was designed to be process-based, simple, and straightforward; the aim being to carry those properties forward into the view-based abstraction model and methodology while pushing any potential complexity into the implementation.

The six coupling steps developed for this dissertation are given below in Figure 3-1. They correspond to the coupling process illustrated in Figure 1-6, in which a programmer is coupling one module (M_1) to another (M_2).

I. *Boundary Identification* - Identify the abstraction boundaries in the code

II. *Precondition Development* - Develop the preconditions under which the desired coupling is valid

III. *Code Analysis* - Analyze the code, to gather the program properties needed to validate the preconditions

IV. *Modification Development* - Develop code modifications for changing the uncoupled code into the desired code

V. *Coupling Production* - If the preconditions are valid, carry out the code modifications to produce coupled code

VI. *Recoupling* - Whenever the coupling code or the uncoupled code is modified in the future, redo Steps I-VI to produce a new, valid piece of coupled code (or alternatively but equivalently, whenever the coupling code or the coupled code is modified, undo a previously applied coupling if it is no longer valid)

Figure 3-1 - Implementation-Coupling Steps

Two critical ideas were drawn from the implementation-coupling steps. The first is that implementation coupling does not necessarily preclude modularity in the eyes of future programmers. A coupling $Impl(M_1) \mapsto Impl(M_2)$ prescribes only that an invalidating implementation, $Impl'$, of M_2 exists. This means that modularity is precluded *only when the recoupling step is not carried out*. If the recoupling step is always carried out, an existing implementation coupling cannot persist through future program modifications unless its preconditions remain valid. This idea is further amplified below, after the implementation-coupling steps have been discussed.

The second critical idea is that by providing an imperative language for expressing the boundary identification, precondition development, and modification development steps, the code analysis, coupling production, and recoupling steps can be easily automated. This automation restores modularity in the presence of implementation coupling, since the computer can then be relied upon to carry out the recoupling step. This idea is also amplified below, after the implementation-coupling steps have been discussed.

3.2 The Implementation-Coupling Steps

The six implementation-coupling steps are presented in the context of a programmer who has a particular coupling in mind (i.e., $Impl(M) \mapsto Impl(M_2)$). Either this programmer or a future programmer must maintain modularity by redoing the implementation-coupling steps when the original, uncoupled code or the coupling code is modified.

3.2.1 Step I - Boundary Identification

The purpose of Step I is to break up an implementation into pieces that are easier to reason about. This typically means generating abstraction boundaries around groups of expressions that implement interfaces (i.e, generating boundaries around modules). For the dispatch example, there are two such groups of code. The first is `dispatch-expression` and its associated `process-<exp>` expressions (i.e., Figure 1-2). The second is `exp-type` and the expressions that compute expression types (i.e., Figure 1-3). These boundaries help to distinguish true implementation couplings from those modifications that would otherwise be allowable under black-box abstraction. The boundaries will also be used in later steps, to identify which groups of expressions must be analyzed to validate preconditions.

3.2.2 Step II - Precondition Development

The purpose of Step II is to develop criteria for determining whether the desired program implementation does not introduce bugs into the application. This typically involves developing and then specifying a set of preconditions under which a coupling is valid. These preconditions imply that $ValidCoupling(< Int(M), Impl' >, M_2)$ is true, where $Impl'$ is the desired, coupled code. Preconditions must always be correctness preserving, but not necessarily semantics preserving.

Precondition development can be one of the most difficult steps in the implementation-coupling process. To better understand why, let us order preconditions along a spectrum from weak to strong. Suppose we have two preconditions, P_a and P_b, which can only be satisfied if a desired implementation coupling is valid (i.e., correctness preserving) with respect to a program of size less than a large number of characters, n.[‡] Suppose $S(P_a)$ is the set of all programs for which P_a is satisfied, and $S(P_b)$ is the set of all programs for which P_b is satisfied. We say that P_a is *strictly weaker* than P_b if $|S(P_a)| > |S(P_b)|$. That is, P_a is valid for more programs than P_b. Weaker preconditions are therefore preferable to stronger preconditions, as weaker preconditions can be satisfied by a greater number of programs for which a desired coupling is valid. As is explored below, however, weaker preconditions can be more difficult to express.

A strong precondition can be relatively straightforward to develop compared to a weaker precondition. A strong precondition can even be expressed as a code template in a pattern language. For instance, in the dispatch example, one could develop two simple strong preconditions for Step II. The first precondition, P_1, is that the body of `dispatch-expression` matches the following piece of code:

[‡] The size requirement forces the sets subsequently discussed to be finite, so that their sizes can be compared. n should be some huge finite number, such as 2^{500}, so that all existing programs are smaller than this number of characters.

```
(lambda (exp)
 (cond ((type? 'literal       exp) (process-literal       exp))
       ((type? 'java-name     exp) (process-java-name     exp))
       ((type? 'new           exp) (process-new           exp))
       ((type? 'dot           exp) (process-dot           exp))
       ((type? 'call          exp) (process-call          exp))
       ((type? 'cast          exp) (process-cast          exp))
       ((type? 'instanceof    exp) (process-instanceof    exp))
       ((type? 'built-in-expr exp) (process-built-in-expr exp))
       ((type? 'assignment    exp) (process-assignment    exp))
       (else (error "Unknown Type: " exp))))
```

The second strong precondition, P_2, is that the body of `type?` matches:

```
(lambda (type-keyword exp)
 (eq? type-keyword (exp-type exp)))
```

These two strong preconditions validate the transformation to the target code in Figure 1-4.

The tradeoff for using a strong precondition, however, is fragility. A *fragile* precondition is one in that is likely to be invalidated by minor semantics-preserving source-code modifications. A weaker precondition is less likely to be invalidated by semantics-preserving source-code modifications. For example, the precondition P_1 given above is fragile because it will be invalid if `type?` in `(type? 'literal exp)` is replaced by any function call that returns `type?`'s value (such as `((lambda () type?))`) or even if `exp` is renamed. In both cases, the coupling is still valid, even though in P_1, a strong precondition is not. The important point is that fragile preconditions can hamper maintainability by disallowing future program modifications that do not invalidate an implementation coupling but do not satisfy the coupling's preconditions.

Weak preconditions, on the other hand, are not as easy to develop. Unlike strong preconditions, however, weak preconditions can be made less fragile. By combining weak preconditions with the strong ones, for example, some of the previously identified fragility can be removed. P_1', below, illustrates this point. P_1' is a precondition on `dispatch-expression`:

- `dispatch-expression`'s value is a procedure created by a `lambda` expression whose body is a `cond` clause such that:
  - Each predicate is the following combination: `(<type?> <symbol> <exp>)` where `<type?>` is an expression that returns the value of `type?`, `<symbol>` is a literal symbol, and `<exp>` references `dispatch-expression`'s parameter
  - Each consequent clause is the following combination: `(<process-exp> <exp>)` where `<process-exp>` is a procedure and `<exp>` references `dispatch-expression`'s parameter
- The variable `exp` is not mutated via `set!`
- Predefined primitives are not re-defined

Similarly, the syntactic/semantic precondition P_2' can be on `type?` can be:

- `type?`'s `lambda`'s body is of either the form `(eq? <exp-type> <exp>)` or the form `(eq? <exp> <exp-type>)`where `<exp-type>` is any expression and `<exp>` is `type?`'s second formal parameter

P_1' is less fragile than P_1 in three significant ways. The first is that P_1' remains valid if `type?` is replaced by any expression returning `type?`'s value. The second is that P_1' remains valid if the variable `exp` is renamed. The third is that P_1' remains valid if `cond` clauses are added or removed. This is how semantic preconditions can enhance maintainability; by remaining valid in the presence of a greater variety of semantics-preserving (or even non-semantics-preserving) modifications.

One can produce a much weaker and less fragile precondition than P_1'. For instance, P_1' will fail if `(type? 'cast exp)` is replaced by `(type-cast? exp)`, where `type-cast?` performs the semantic equivalent of `(type? 'cast exp)`.[§] Instead of requiring a `cond` clause predicate to have the form `(<type?> <symbol> <exp>)`, a less fragile precondition, P_1", can simply include the condition that each `cond` clause predicate must return the result of calling `type?` on one symbol and on the decaf-java expression passed to `dispatch-expression`. This reduced fragility, however, is not necessarily free. The price to be paid can be in the form of additional computational resources. This is explained below during the discussion of the code analysis step.

For more complicated pieces of code, determining whether a specified precondition is the weakest precondition is non-trivial. This is the crux of why developing weak preconditions can be difficult: making preconditions less fragile generally requires more effort and more careful thought. Personal experience with the examples in Chapter 5 confirms this. It is also worthy noting that programmers generally have little need to develop less fragile preconditions unless they are interested in documenting the preconditions in order to preserve modularity in the presence of future code modifications. This can help explain why programmers are not as motivated to document implementation couplings they install.

3.2.3 Step III - Code Analysis

At some point, the preconditions must be validated. In the dispatch example, P_1 can be validated by inspection: the operator in `(type? 'cast exp)` does reference `type?`'s value. This is a trivial precondition to validate; strong preconditions tend to fall into this classification. Some preconditions, however, cannot be easily validated by inspection. These are non-trivial preconditions, and weaker preconditions such as P_1" usually fall into this classification. Preconditions typically become non-trivial when they must be validated against non-local program properties such as call graphs or data-flow analyses. For example, suppose `(type? 'cast exp)` was replaced

[§] For example, `(define (type-cast? exp) (type? 'cast exp)))`

by `(type-cast? exp)` (in the Figure 1-2 code). To validate P_1", the expression primitively creating `type-cast?`'s return value would have to be examined. This expression could be found via a data-flow analysis.

This is what gets done in the code analysis step: determining the program properties necessary to test the Step II preconditions. These properties can include control-flow, data-flow, naming, and side-effect information. During an implementation coupling, the programmer generally computes this non-local information manually, by inspecting the files containing the code itself.

As previously mentioned, the code analysis step is where less fragile preconditions can become more costly. While the fragile P_1 precondition needs only local, syntactic information from the code analysis step, P_1" needs a non-local data-flow analysis. In addition, automated analyses such as data flow and control flow are not always precise. While this is not the case for the dispatch example, Chapter 5 examines examples where this issue must be confronted.

3.2.4 Step IV - Modification Development

The purpose of the modification development step is to determine how to modify the uncoupled piece of code, $Impl(M)$, into $Impl'$,such that $ValidImpl(Int(M_1), Impl', Dep(M_1))$ is true, where $Impl'$ expresses the desired coupling. In the dispatch example, this means deciding how to transform the code in Figure 1-2 into the code in Figure 1-4. Programmers sometimes do this "on the fly," using a program text editor. This is mostly easy to do with the `dispatch-expression` example by collecting the type symbols and the `process-<exp>` expressions into a hash table. The somewhat more difficult part is deciding which expression in `type?`'s body returns the type symbol given a decaf-java expression.

One measure for comparing the quality of two modifications is their *scope*, or the extent of the expressions that must be modified or that the modification explicitly depends upon. In the dispatch example, `dispatch-expression`'s body is modified, and the modification depends on the body of `type?`. Thus, the scope of the modification is limited to the bodies of `dispatch-expression` and `type?`.

3.2.5 Step V - Coupling Production

The coupling production step is the process of actually carrying out the modifications (from the modification development step) if the preconditions can be validated against the program properties from the code analysis step, with respect to the abstraction boundaries from Step I. In the dispatch example, the programmer manually carries out all these steps to produce the coupled `dispatch-expression` code from the uncoupled code. The coupling production step thus establishes the implementation coupling.

3.2.6 Step VI - Recoupling

Once an implementation coupling exists, the recoupling step ensures that modularity is maintained. It is not practical to guarantee, however, that the programmer or a future programmer will carry out a recoupling for each implementation coupling for each future modification to the uncoupled code or to the coupling code. This is why black-box abstraction cannot guarantee modularity in the presence of implementation couplings: black-box abstraction does not guarantee that only valid couplings will persist through all future program modifications.

If a programmer were, in fact, interested in guaranteeing modularity after having performed a coupling via Steps I-V, the recoupling step would be easier to perform if the preconditions were strong, the code analyses were easy to compute, and the modifications were minimal in scope. The same holds true for an automatic way of guaranteeing modularity, which leads to the two critical ideas forming the basis of view-based abstraction.

3.3 Two Critical Ideas

The recoupling step leads to a critical idea forming the fundamental basis of view-based abstraction. The idea is that, given the six-step implementation-coupling model, *modularity is precluded by an implementation coupling only if that coupling is allowed to persist through future code modifications when the coupling is no longer valid.*

For example, suppose after `dispatch-expression` was coupled to `type?`'s implementation, `type?`'s implementation was modified (as in Figure 1-5) to be:

```
(define (type? type-keyword expression)
  (eq? type-keyword (fast-exp-type? expression)))
```

Now, instead of `exp-type`, the procedure `fast-exp-type` is used to find type symbols. In this case, `type?` still maintains a valid interface, but the coupled code in Figure 1-5 does not. At this point, there are two possibilities. The first possibility is what normally happens under black-box abstraction: modularity was violated and `dispatch-expression`'s coupled implementation now has a bug due to an invalid implementation coupling. The second possibility is to instead redo the coupling (e.g., via Steps I-V) to ensure that either the coupling does not persist or that `dispatch-expression`'s newly coupled implementation uses `fast-exp-type` instead of `exp-type`. Either eliminating the coupling or recoupling using `fast-exp-type` maintains the validity of `dispatch-expression`'s interface, meaning that modularity will be preserved. The first critical idea is that this second possibility is what actually happens in the recoupling step, thereby maintaining modularity.

From this idea, one can conclude that being able to *automatically* carry out the recoupling step would provide the guarantee needed to maintain modularity. This is because the recoupling step ensures that a coupling persists through future code modifications only if the coupling is valid in the context of those future modifications. A maintenance programmer would thus be free to modularly modify a program in the presence of implementation couplings, leaving the work of ensuring validity to the computer.

The real power of this six-step implementation-coupling model and the first critical idea drawn from it is that together, they allow the implementation-coupling problem to be reduced to the problem of automating the recoupling step. This is now a tangible problem, and leads to the second critical idea underlying view-based abstraction: the recoupling step can be automated if the code analysis and coupling production steps are automated, and these steps can be automated if the boundary identification, precondition development, and modification development steps are expressed in an imperative programming language. This *coupling language* for automating and expressing various steps draws inspiration from the established research efforts described in Chapter 2 and is formally presented in Chapter 4.

These two critical ideas led to the development of the view-based abstraction model and methodology. This dissertation therefore addresses the implementation-coupling problem by developing this model and methodology for automating the recoupling step.

3.4 View-Based Abstraction

The view-based abstraction model is based on black-box abstraction. View-based abstraction, however, permits implementation coupling via the process given by Steps I-VI. To accomplish this, view-based abstraction provides a coupling language for expressing the results of boundary identification, precondition development, modification development and couplers (which can then invoke code analyses and coupling productions). View-based abstraction also specifies a component called the view invalidation/recoupler to carry out the recoupling step. View-based abstraction is backwards compatible with black-box abstraction, is language independent, and is incremental. Being a straight-forward extension of black-box abstraction also makes view-based abstraction easier to use and understand.

Below, the view-based abstraction model and methodology are summarized at a high level. The model and methodology are then presented in more detail, are formalized, and are related to the dispatch example.

3.5 Summary of the View-Based Abstraction Model

The black-box abstraction model (illustrated in Figure 1-1) relates three kinds of components: modules, interfaces, and implementations. Modules can be combined via interface coupling to form new compound modules, and implementation coupling is prohibited.

By comparison, the view-based abstraction model (see Figure 1-7) extends black-box abstraction with two components, one mechanism, and a coupling language: views and contexts, a view invalidation/recoupler, and a coupling language. Views hold the results of code analysis. A view is a mapping from one or more program expressions to properties about those expressions. A naming view, for example, includes a mapping from formal parameters (i.e., a "def") to variable references (i.e., a "use"). Views are invoked as needed to test preconditions, where "invoking a view" means computing the view's mapping. A context is a group of program expressions and its associated views. Contexts typically correspond to black-box abstraction modules, and are used to limit the extent of preconditions, modifications, and views. When a view is invoked on a context, for example, the view analyzes nothing beyond the expressions in the context.

In the view-based abstraction model, programmers do not perform implementation couplings; couplers do. Couplers are written in the coupling language: an imperative, program-transformation-based programming language. The coupling language is used to express the results of boundary identification, precondition development, and modification development. These are combined to form couplers that perform the coupling production step. Couplers create contexts, ensure the proper views are invoked, test preconditions, and couple code via Step IV modifications when those preconditions are valid.

The view invalidation/recoupler carries out the recoupling step. Each time a coupler performs an implementation coupling, it registers the coupler with the view invalidation/recoupler. When a coupled context's views or a coupling context's views become invalid (e.g., the context's expressions are modified), the view invalidation/recoupler re-invokes the coupler that created the coupled context. The coupler recouples the context only if the corresponding preconditions are still valid.

3.6 Summary of View-Based Abstraction Methodology

The view-based abstraction model is designed to support implementation coupling via Steps I-VI. The view-based abstraction methodology supplements Steps I-VI with Steps i-vi as given below:

I. Boundary Identification
   i) *Context Identification* - Set up *contexts* corresponding to these abstraction boundaries

II. Precondition Development
   ii) *Predicate Development* - Write *predicates* expressing the preconditions

III. Code Analysis
   iii) *View Development* - Ensure the *views* needed to validate the predicates are available

IV. Modification Development
   iv) *Action Development* - Write *actions* expressing the modifications

V. Coupling Production
   v) *Coupler Production* - Create a *coupler* from the predicates and actions, invoke the coupler, and register the coupling with the view invalidation/recoupler if successful

VI. Recoupling
   vi) *Invalidation/Recoupling* - Invoke the *view invalidation/recoupler*

The model and the methodology presented in this overview are described in the sections that follow.

3.7 View-Based Abstraction Components

3.7.1 Views

Views fit into the implementation coupling process at the code analysis step. A view computes, contains and manages semantic properties of a group of program expressions. These semantic properties can be used to test a precondition's validity. Formally, a view is a mapping from program expressions (or other objects) to their semantic properties:

$$V = \{((exp|obj) \rightarrow computed\text{–}property)^*\}$$
$$computed\text{–}property = obj$$

The meaning of *computed-property* with respect to *exp* depends on the particular view, *V*. For instance, the alpha view (detailed in Section 4.6.2) maps formal parameters to variable references and to variable mutations. Variable references and variable mutations are also mapped back to formal parameters. Additionally, the alpha view maps a group of expressions to the list of unbound variables in that group.

A different view used by the dispatch example is the liar view, based on the LIAR data-flow analysis algorithm.[79] This view maps expressions to data-flow properties. Figure 3-2 is a graphical representation of the mapping for `repeated`'s formal parameter `f`, with respect to the group of given program expressions. It shows that `f` may be assigned one of two procedure val-

ues or any values named by the variable `proc`. It also shows that `f`'s value may be applied to the value of the variable `current`. This figure does not illustrate the fact that as a parameter to a top-level procedure, `f` may be bound to any denotable value.

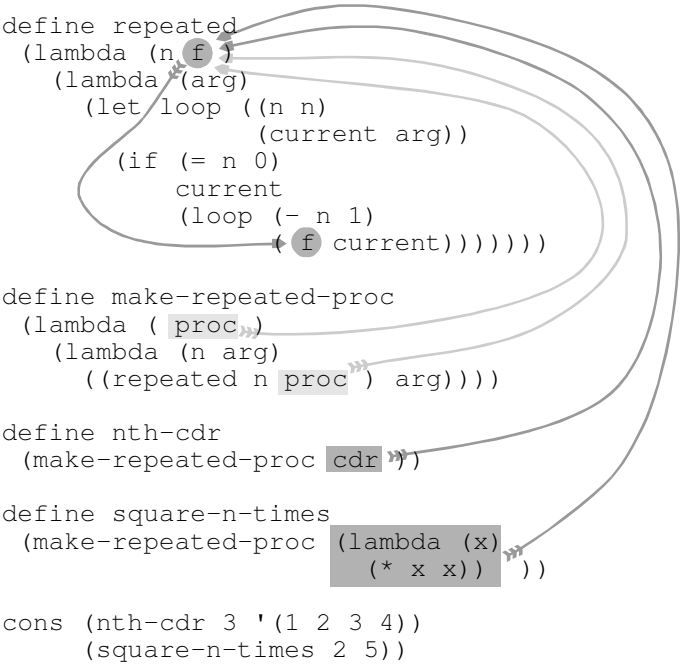

Figure 3-2 - Partial Data Flow Analysis on `f`

Each implemented view provides an interface containing functions for creating, invoking, retrieving, merging, and modifying the view's mapping. For instance, in the dispatch example, `ensure-alpha-view-computed!` is the function invoked to compute the alpha view (on a context, described next), and `var-binding->mutation` is the function used to find all the `set!` expressions that mutate `dispatch-expression`'s formal parameter, `exp`. The view interface specification is presented in Chapter 4.

By abstracting away the details of computing common collections of program properties, views allow preconditions to be developed at a clearer and higher level of understanding. By using views, preconditions are freed from having to specify the details of how a semantic program property is computed. For instance, the precondition P_1' says, "the variable `exp` is not mutated via `set!`". This precondition does not specify *how* the mutation properties of `exp` are determined. The way in which those properties are determined has instead been delegated to a code analysis naming view. This separation between semantic preconditions and semantic program properties implies that P_1' does not have to commit to a particular algorithm for determining `exp`'s mutation sites. The actual algorithm will instead depend on the implementation of the particular view chosen to provide the information.

In view-based abstraction, couplings persist based on the validity of the views used to validate their preconditions. A view is valid so long as the program expressions upon which the view information is based are not modified. When these expressions are modified, the corresponding views become invalid. Preconditions that depend on the validity of the view information may then become invalid themselves. These invalid preconditions can then invalidate existing couplings. This is what happens when the Figure 1-5 invalidating implementation is substituted for the Figure 1-3 coupling code. This invalidation cascade can be reversed by updating the view, re-testing the preconditions, and re-coupling the code if the preconditions are valid.

3.7.2 Contexts

Contexts correspond to the boundary abstraction groupings of program expressions. In view-based abstraction, contexts are also used to limit the number of expressions that a view must analyze. This implies that the program properties contained in a view are with respect to a given context. Contexts can be formalized as:

$C =< \{exp^*\}, \{V^*\} >$

Each context is composed of a set of program expressions and a set of views containing properties of those expressions. As the equation above shows, a view is always associated with a context. A view has no meaning outside of its associated context's expressions.

The dispatch example has two clear contexts: the context containing `dispatch-expression` and the `process-<exp>` definitions, and the context containing `type?`, `exp-type`, and the other expressions for determining a decaf-java expression's type. These two contexts line up with the dispatch example's abstraction boundaries, and are stored in two files. The function `make-context-from-files` is actually used in the dispatch example to create the two contexts.

Contexts can also be combined into larger contexts. Since a view can analyze the expressions in exactly one context, merging is an important way to analyze the union of expressions in several contexts. In addition, when contexts are combined, their views are too. This means that views must provide a merge function that the context-combining functions (i.e., `merge-contexts`, `merge-contexts!`) can invoke.

3.7.3 Coupling Language

The coupling language is an imperative language based on program transformations. The coupling language layers five main constructs over an imperative base language (such as Scheme[49], for this dissertation). These four constructs are: predicates, actions, dispatchers, rules, and couplers. Preconditions are expressed as predicates, program modifications as actions, conditional rewrite rules as dispatchers, combinations of constructs as rules, and implementation couplings as couplers. These constructs are described below.

3.7.3.1 Predicates

A predicate is an imperative expression of a Step II precondition. A predicate may also use the program properties stored in a view to determine whether a particular set of expressions in the view's associated context satisfy a precondition. Predicates that use views that conservatively approximate program properties will correspondingly be conservative approximations to the preconditions they express. Predicates have the following signature:

$Pred$: $Context \times exp^* \rightarrow bool$

$bool$: $true | false$

A predicate returns true if the precondition it implements is valid, and false otherwise. A sample predicate related to the dispatch example is given below. This predicate returns a true value when the given variable binding is not mutated via `set!` in a given context. The code for this predicate is further explained in Chapter 4.

```
(define predicate/no-mutations
   (make-explained-predicate (lambda (vform context variable-binding)
                               (null? (var-binding->mutations variable-binding context)))
     "identifying variable bindings with no mutations"))
```

3.7.3.2 Actions

An action is an imperative expression of a Step IV program modification. Actions unconditionally modify a context's expressions. Actions are also free to use views for locating expressions to be modified, or for gathering sub-expressions to be used in a modification. Actions have the following signature:

$Action$: $Context \times exp^* \rightarrow bool$, modifies $Context$

An action returns true if the modification was successful and false otherwise. The following action renames a variable and its references to `foo-bar`. It depends on a valid alpha view and on the alpha view function `var-binding->var-refs` to gather a variable binding's references. Action implementation is further discussed in Chapter 4.

```
(define action/rename-foo-bar
  (make-explained-action (lambda (vform context variable-binding)
                           (for-each (lambda (var) (replace var 'foo-bar))
                                     (var-binding->var-refs variable-binding context))
                           (replace variable-binding 'foo-bar)
                           #t)
  "renaming variable references and their binding to foo-bar"))
```

3.7.3.3 Dispatchers

Dispatchers correspond to the simplest kinds of couplers. A dispatcher is a simple combination of a predicate and an action. A dispatcher first runs a predicate. If the predicate is true, the dispatcher runs the action. Dispatchers have the following signature:

$Dispatcher$: $Context \times exp^* \rightarrow bool$, can modify $Context$

A dispatcher returns true if successful, and false otherwise. Dispatchers are similar to what are commonly referred to as conditional rewrite rules, as illustrated in Figure 3-3. Dispatchers differ from typical rewrite rules in that dispatchers permit non-semantics-preserving transformations

(via actions), can depend upon non-local program properties (via views), and, as is later discussed, can rely upon user interaction.

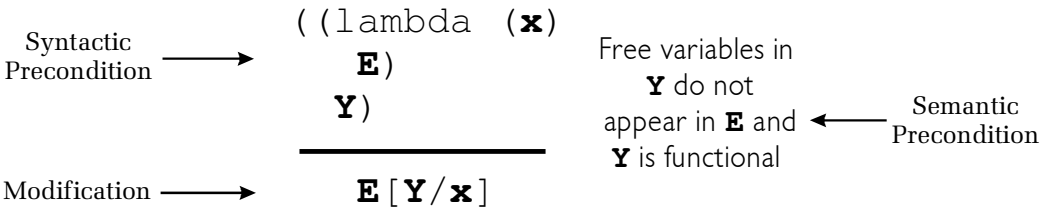

Figure 3-3 - Conditional Rewrite Rule

The following code combines the predicate and action given above to produce a dispatcher that renames a given variable to `foo-bar` if the variable is not mutated via `set!` in a given context:

```
(define dispatcher/rename-to-foo-bar
   (make-simple-dispatcher predicate/no-mutations action/rename-foo-bar))
```

3.7.3.4 Rules

Rules are combinations of predicates, actions, dispatchers, or other rules. In fact, a dispatcher is just a simple, but common kind of rule. Rules are used to build couplers, which are responsible for producing one or more complete implementation couplings. A complete coupling means creating the contexts, ensuring that the proper views are computed on the right contexts, running dispatchers on the right program expressions and contexts, and registering couplings with the view invalidation/recoupler. Rules have the following signature:

Rule: $Context \times exp^* \rightarrow bool$, can modify *Context*

A rule is created from a control structure and a set of coupling constructs, usually dispatchers or other rules. The control structure is a function that takes a set of coupling constructs and returns the rule itself. For example, the control structure below creates a context, ensures the alpha view is computed on that context, invokes all of its passed-in coupling constructs, and registers a successful coupling with the view invalidation/recoupler. Most control structures will take this form. The rule that follows is created using this control structure. Writing and understanding code like this is further discussed in Chapter 4.

```
(define (make-foo-bar-control-structure file1)
   (lambda (vforms)
     (lambda (vform context exps)
       (let ((context (make-context-from-file file1)))
         (ensure-alpha-view-computed! context)
         (if (for-all? vforms
                (lambda (vform) (do-vform vform context exps)))
             (begin (register-coupling! context vform)
                    context)
             #f)))))
```

```
(define rule/rename-to-foo-bar
   (combine-vforms (make-foo-bar-control-structure "test-file.scm")
                   (verbose-expl-combiner "renaming variables to foo-bar")
                   (list dispatcher/rename-to-foo-bar)))
```

3.7.3.5 Couplers

Now that rules and contexts have been discussed, it is possible to more precisely define what a view-based abstraction coupling is. A coupling exists when a rule modifies a context's expressions. This rule is the coupler. The original context is the uncoupled context, whereas the modified context is the coupled context. If the coupler's preconditions depended on a context's views for validation, this context is called a coupling context.

In the dispatch example, the uncoupled context contains the expressions in Figure 1-2. The coupled context contains the expressions in Figure 1-4. The context in Figure 1-3 containing the definition for `type?` is a coupling context.

3.7.3.6 View Invalidation/Recoupler

The view invalidation/recoupler is responsible for performing the recoupling step. For each coupling, the view invalidation/recoupler maintains enough information to test for the coupling's validity and to re-invoke the coupler if necessary. Re-invoking the coupler is necessary when the uncoupled context or coupling context is modified. In the former case, modifications to the uncoupled context must be reflected in the coupled code. In the dispatch example, for instance, adding a `cond` clause for a new decaf-java expression type would require recoupling. In the latter case, modifying the coupling context potentially renders its views invalid with respect to the coupling. Modifying `type?`'s implementation is an example of this kind of modification. In both cases, re-invoking the coupler is necessary.

When invoked, the view invalidation/recoupler goes through the couplers in its registry to decide which ones need to be re-invoked. Associated with each coupler is enough information to determine whether an invalidation has occurred. This information can take the form of the names and modification times of the files used to store the uncoupled context and coupling context expressions.

More formally, the view invalidation/recoupler identifies any couplings making *ValidProgram*(*P*) false, where P contains the modified context expressions. After the view invalidation/recoupler runs,*ValidProgram*(*P*) will be true by virtue of the view invalidation/recoupler having re-invoked the couplers that produced the invalid couplings.

3.8 View-Based Abstraction Methodology

The view-based abstraction model provides a set of components designed to support the implementation coupling process given in Steps I-VI. The view-based abstraction methodology supplements Steps I-VI with Steps i-vi, given above in Section 3.6. Each of these supplemental steps is discussed below.

3.8.1 Step i - Context Identification

In the context identification step, the program expressions are grouped into contexts. This grouping is important, as views, coupling constructs, and the view invalidation/recoupler all perform their functions with respect to contexts. Since contexts can be combined, smaller contexts are a good starting point. Contexts are usually created by a coupler's control structure, and then passed on to the appropriate dispatchers and rules.

As discussed earlier, the dispatch example has two contexts: one containing `dispatch-expression` and the `process-<exp>` definitions, and a second context containing `type?`, `exp-type`, and the other expressions for determining a decaf-java expression's type. These contexts are created by the coupler's control structure from the files containing the respective expressions.

3.8.2 Step ii - Predicate Development

In this step, predicates expressing the Step II preconditions are implemented using the imperative coupling language. Predicates are free to invoke view retrieval functions to extract properties about an expression in a context. For the dispatch example, P_1' and P_2' suffice as preconditions, and must therefore be expressed as predicates in the predicate development step. For P_1', one predicate is needed to ensure primitives are not redefined, one is needed to ensure that `dispatch-expression`'s formal parameter is not mutated via `set!`, one is needed to check `cond` clause predicates, and one is needed to check `cond` clause consequents. One predicate is needed to validate P_2'.

3.8.3 Step iii - View Development

In the view development step, one must ensure that any views required to validate the developed predicates are implemented and available. The required views can be determined by inspecting the predicates for the view retrieval functions they call (e.g., `var-binding->mutations`) or by a predicate's specification. If a desired view is not available, it must be implemented as discussed later in Chapter 4.

Given the description of P_1' and P_2' above, it is clear that the predicates will need two views: a naming view and a data-flow analysis view. The naming view is used to find possible muta-

tions among other things, and the data-flow view is used to ensure that in the `cond` predicate clauses `(<type?> <symbol> <exp>)`, the operator `<type?>` only references the value of `type?`. These views are provided by default in the view-based abstraction implementation discussed in Chapter 4. Chapter 4 also demonstrates the implementation of a simple view. For the dispatch example, it suffices for now to keep track of which default views are required.

3.8.4 Step iv - Action Development

For the action development step, the modification developed in the modification development step is implemented as an action in the coupling language. Like predicates, actions are free to invoke view retrieval functions. Unlike predicates, actions use views to locate the expressions to be modified, or to gather code to be used in a modification. Since actions are imperative, they directly modify the original code into the desired code.

In the dispatch example, it is necessary to invoke a view function while modifying `dispatch-expression` because the action needs part of `type?`'s code (specifically, the call to `exp-type`). One action is necessary for the dispatch example; the action that takes the original `dispatch-expression` code into the coupled version.

3.8.5 Step v - Coupler Development

Under view-based abstraction, the coupler development step involves combining the predicates and actions into dispatchers, then combining the dispatchers into a single, main rule (or coupler). The reason a single rule is necessary is that the view invalidation/recoupler invokes a single rule when an invalid coupling is detected. This condition does not result in any loss of generality, as rules are easily combined.

A coupler's control structure creates any necessary contexts, ensures the views identified in Step iii are computed, invokes other rules and dispatchers on the appropriate set of expressions and contexts, and registers any successful couplings with the view invalidation/recoupler.

3.8.6 Step vi - Invalidation/Recoupling

This is the most straightforward of the steps. The view invalidation/recoupler must be an invokable function, and performing the invalidation/recoupling step means invoking this function after the code base has been modified, but before any coupled code is run. This function can be invoked eagerly (immediately after a code modification), lazily (before any modified code is run), regularly (by a cron job or by a programming environment) or manually (by the programmer).

Chapter 4

ViewForm

There are a variety of ways to implement view-based abstraction as described in Chapter 3. My approach is to relate the view-based abstraction implementation to the six-step coupling process. I also strive to simplify the interfaces typically used by the programmer while providing more complex and more powerful interfaces for experienced programmers. The result is ViewForm, developed to experiment with the view-based abstraction model and methodology. ViewForm is an imperative, transformation-based language layered over Scheme[49]. ViewForm performs source-to-source transformations on Scheme code augmented with various MIT Scheme constructs. ViewForm also introduces a novel construct, the *vform*. Vforms are combinable, delegation-based constructs that operate on program expressions, with respect to a context. Vforms are used in the construction of predicates, actions, dispatchers, rules, couplers, and views.

In addition to the view-based abstraction coupling constructs, ViewForm implements various default views, as well as a simple but fully functioning view invalidation/recoupler. ViewForm uses complexity layering to make more common uses of view-based abstraction easier to implement. ViewForm maintains backwards compatibility with current software engineering practice, provides incrementality, and demonstrates view-based abstraction's viability. This chapter's goal is to describe ViewForm to the point that a reader could begin implementing couplings via Steps i-vi.

4.1 ViewForm Overview Scenario

Figure 4-1 illustrates a high-level overview of ViewForm. This scenario illustrates how a coupling is created using ViewForm, and how the view-based abstraction components fit together inside ViewForm. The coupler in the figure calls on ViewForm to take the source code and parse it into *viewcode*, ViewForm's internal program representation. The coupler then groups the viewcode expressions into contexts, and proceeds to invoke selected views on selected contexts. At this point, the coupler can apply various dispatchers to specific expressions

with respect to a context. A dispatcher's predicate can use the view information to validate a precondition, and a dispatcher's action can modify a context's expressions. Once the coupler has finished invoking the dispatchers, it can register the coupling (if any) with the view invalidation/recoupler (not shown). The coupled contexts can be output by calling the unparser on the appropriate contexts. The output of ViewForm's unparser is suitable for direct interpretation or compilation. It is important to note that the output is not intended to replace the original code. Rather, ViewForm should be thought of as a preprocessor for the code; its output is subject to automatic regeneration. The "explanations" in the figure represent components that can be invoked to provide a description of the associated construct's functionality.

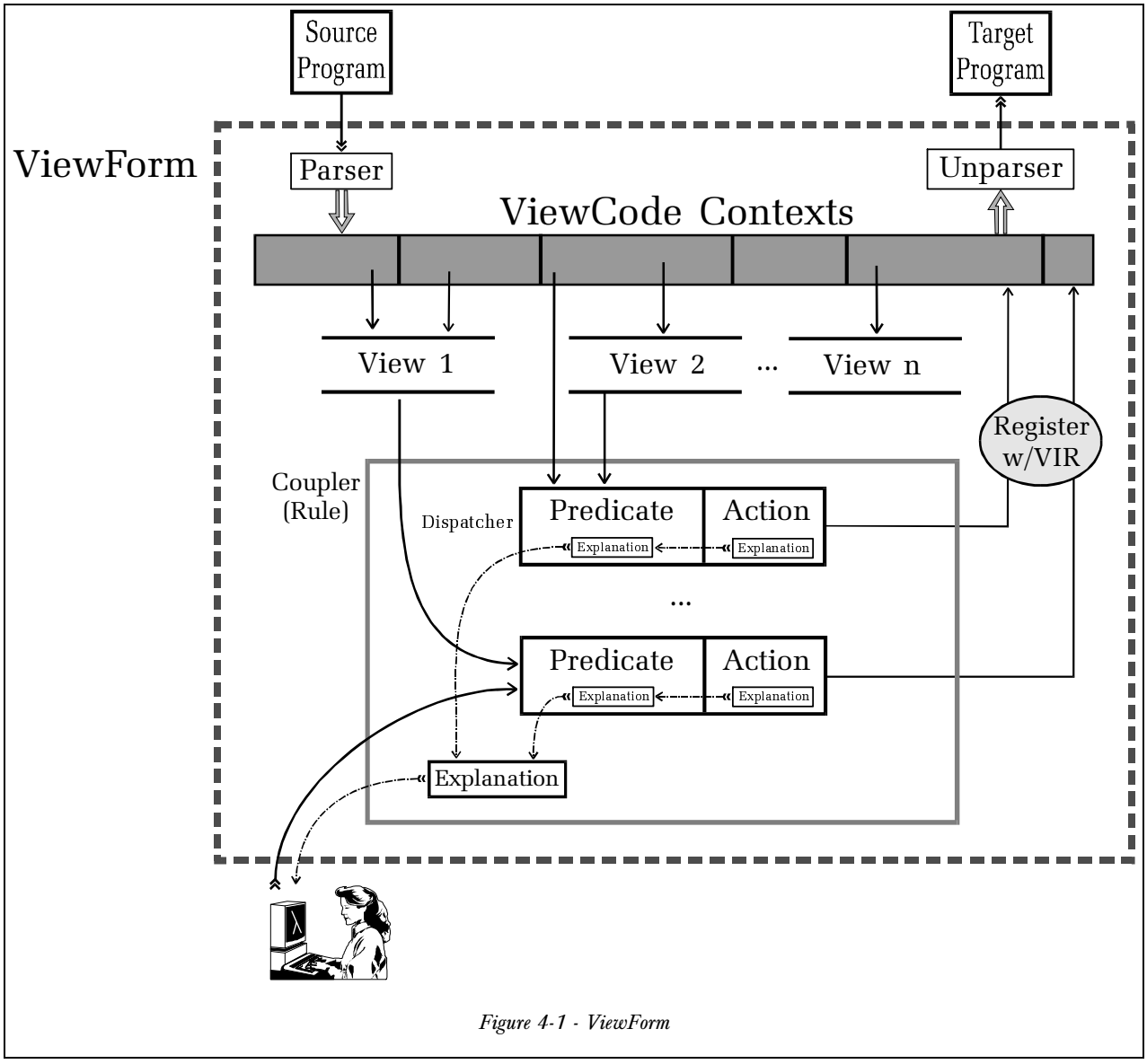

Figure 4-1 - ViewForm

The ViewForm view invalidation/recoupler relies upon file modification times to determine whether a context's original expressions have been modified.[**] When a coupling is registered, the view invalidation/recoupler notes every coupling context's and coupled context's corresponding file modification times. When invoked, the view invalidation/recoupler checks through its registry for files that have been modified since it was last invoked. If any are found, the view invalidation/recoupler assumes that views computed from those contexts are invalid. The couplers that originally coupled the contexts are then re-invoked to produce valid, freshly coupled code.

In the following sections, the more interesting and relevant ViewForm interface functions are documented and described (the remainder are given in Appendix A). First, viewcode and contexts are described, followed by ViewForm's coupling constructs. ViewForm's default views and the view invalidation/recoupler are documented and described afterwards. The ViewForm code used to couple the dispatch example is then presented. Throughout this section, referring back to Section 3.7 is recommended as a way to relate the ViewForm constructs back to their corresponding view-based abstraction components.

4.2 Viewcode

The ViewForm parser builds viewcode out of Scheme objects. These Scheme objects are created from Scheme expressions, passed in standard `quote`d form to functions like `make-context` or created from files containing Scheme code.[††] When the viewcode is ready for output, it is unparsed back into Scheme code that can be interpreted or compiled. The parser is built into the implementation of every ViewForm function that takes source-level input.

The parser performs some simple manipulations on the input before rendering the viewcode. Some manipulations bootstrap the transformation process, some ensure identity for every viewcode expression, and some canonicalize the input. The manipulations strive to keep the viewcode as close to the original source as possible, in order to maintain any prior familiarity the programmer may have had with the code. For example, one such manipulation is desugaring the expression `(define (foo …) …)` into the canonical `(define foo (lambda (…) …)`.

Viewcode's grammar mirrors that of Scheme and as such, viewcode compound expressions can be traversed using the standard pair operations such as `car` and `cdr`. The point where viewcode's grammar differs from that of Scheme is with respect to immutable objects. These objects

---

[**] For industrial-strength view-based abstraction implementations, a cryptographic fingerprint would be more desirable than file modification times.

[††] When reading files, ViewForm simply uses Scheme's `read` function to create the Scheme objects passed to the parser.

have no identity under Scheme. Since ViewForm requires identity for each viewcode expression, ViewForm automatically *encloses* these objects in viewcode *enclosures*. An enclosure gives an object identity within a viewcode expression tree.

The viewcode operations can be invoked through the following interface functions:

```
(delete-exp!     exp)
(replace     old-exp new-exp)
(insert-after    exp new-exp)
(insert-before   exp new-exp)
```

Delete, replace, and insert viewcode expressions. Automatically maintains enclosures. `new-exp` can be a quoted Scheme expression, viewcode, or any combination of the two

`(vc-enc? object)`

Returns `#t` if `object` is a valid viewcode enclosure

`(vc-enc/object vc-enc)`

Returns the Scheme object enclosed by `vc-enc`

`(vc->sexp vc-exp)`

Unparses a viewcode expression into a Scheme object which can be directly evaluated

`(dump-contexts contexts #!optional file-extension)`

Outputs each context in the list `contexts` to a file, using the optional file extension if given. The output filenames themselves are determined from the context origins

For example, in the code below, suppose the variable `disp-exp` refers to the viewcode for `(define dispatch-expression (lambda (exp) …))` in Figure 1-2.

```
(list? disp-exp) ➔ #t
(car disp-exp) ➔ define                          ;an enclosure for define
(vc-enc? (car disp-exp)) ➔ #t
(eq? (car disp-exp) 'define) ➔ #f                ;an enclosure is not a symbol
(eq? (vc-enc/object (car disp-exp)) 'define) ➔ #t  ;an enclosure can hold a symbol
(eq? (vc->sexp (car disp-exp)) 'define) ➔ #t
(replace (car disp-exp) 'define-integrable) ➔ unspecified
(car disp-exp) ➔ define-integrable                ;define was replaced by define-integrable
(vc-enc? (car disp-exp)) ➔ #t                     ;parser automatically enclosed the symbol
(car  (caddr disp-exp)) ➔ lambda                  ;special-form keyword lambda (enclosed)
(cadr (caddr disp-exp)) ➔ (exp)                   ;a list of the lambda's formals
(list? (cadr (caddr disp-exp))) ➔ #t
(define-var/value (lambda/formals disp-exp)) ➔ (exp) ;a nicer way to get the lambda's formals
```

4.3 Contexts

Contexts are groups of viewcode expressions that can then be passed to ViewForm constructs such as views and couplers. In addition to containing a set of expressions, each context conceptually "contains" any views computed on its expressions. That is, each view is associated

with a single context. This means that contexts act as a kind of scoping mechanism for views. When reasoning about views, it is therefore important to keep in mind that the properties they provide are always with respect to a given context.

It is often useful to group expressions into contexts based on an application's abstraction boundary granularity. By creating a context for each source-level module (or sub-module), a conceptual correspondence can be established between a module and a context. For example, the module code in Figure 1-2 would make a good context, as would the code in Figure 1-3. This context to module correspondence can make it easier to reason about contexts because it leverages on the programmer's existing understanding of the application. Even so, viewcode expressions can belong to any number of contexts, to allow context merging and copying.

Contexts also maintain a point of origin specifying where the context's original expressions came from. The origin can be a filename or another context. A point of origin is also maintained for each context expression. The view invalidation/recoupler uses these points of origin to check the validity of each context's views. A context's views are considered invalid when one of its origins has been altered. The points of origin also allow explanations (described in Section 4.7) and users to more easily determine where any given expression originally came from. I found this latter ability quite useful when using a working version of ViewForm to debug newer versions.

The following ViewForm functions manage contexts:

`(make-context exp1 ... expn)`

Returns a context consisting of the viewcode expressions `exp1` ... `expn` in some order

`(context/exps context)`

Returns a list of `context`'s viewcode expressions

```
(add-exps-to-context!   context exps)
(replace-exp-in-context context old-exp new-exp)
(delete-exp-in-context! context exp)
```

Add, replace, and delete context expressions

`(make-context-from-files file1 ... filen)`

Creates a context from the expressions in files `file1` to `filen` and creates the corresponding context and expression origins

`(context/origin context)` **`(set-context/origin! context origin)`**

Returns or sets `context`'s `origin`, respectively

(merge-contexts c1 c2) (merge-contexts! c1 c2)

Returns a new context containing the non-redundant union of the expressions in `c1` and `c2`, merging existing views and origins (the second form destructively modifies `c1`)

(retrieve-view view context)

Returns the view mapping associated with the given context

(set-view! view context view-mapping)

Sets the view mapping for the view associated with the given context

(remove-view! view context)

Removes the view mapping associated with the given context

4.4 ViewForm Coupling Constructs

ViewForm predefines a set of coupling constructs for the six steps discussed in Chapter 3. These include two primitives, predicates and actions, and three combinations, dispatchers, rules, and couplers. These constructs are described below.

4.4.1 Characteristic Vform Interface

The coupling constructs given below refer to a *characteristic vform interface*. This is a procedure signature (i.e., lambda-list signature), shared by the constructs, whose form is:

vform:: delegating-vform x context x vc-exps -> any-Scheme-value

`delegating-vform` is a vform, `context` is a context, and `vc-exps` is a list of viewcode expressions

This interface is discussed later in Section 4.5, after the ViewForm constructs and views have been properly detailed. This is necessary to allow the vform interface discussion to demonstrate how the ViewForm constructs are related to the vform interface. For now, it suffices to say that predicates, actions, dispatchers, rules, and couplers are all vforms.

The coupling constructs also refer to the notion of an explanation. These descriptive constructs are discussed in detail in Section 4.7. While not strictly vforms, explanations also have the characteristic vform interface.

4.4.2 Predicates

A predicate expresses one or more validating preconditions for a particular coupling with respect to a view, a set of viewcode expressions, and a context. Predicates are conceptually similar to composition filters[2], except predicates do their work before run time. Primitive predicates can be constructed as follows:

(make-predicate raw-predicate explanation)

Returns a primitive vform. Both the raw predicate and explanation must have the characteristic vform interface. When invoked, the raw primitive is called, and is expected to return a non-`#f` value if successful. When explained, `explanation` produces an explanation of the predicate's precondition.

Predicates typically use view retrieval functions to selectively acquire useful information about viewcode expressions of interest, then process that information to validate a precondition on those viewcode expressions. This implies that a predicate relying on a program analysis that conservatively approximates program properties will therefore conservatively approximately a precondition.

To express a precondition as a predicate, a programmer must determine which views to use, which view retrieval functions to use, and how to use the retrieved information to compute the validity of the precondition. Suppose, for example, that we wanted to implement a predicate that takes a list of `lambda` expressions and determines whether any of their formal parameters is mutated. The alpha view previously mentioned (and whose interface is discussed later in this chapter) computes lists of mutations. Such a list can be retrieved via the function `var-binding->mutations`. If this list is empty for a given formal, then the formal is not mutated. The predicate below implements this functionality (`make-simple-expl`, a higher-level function for building explanations, is discussed in Section 4.7 and `lambda/formals` is just a descriptive name for `cadr`.)

```
(define predicate/no-mutations
  (make-predicate
    (lambda (vform context lambda-exps)
      (for-all? (append-map lambda/formals lambda-exps)
        (lambda (formal) (null? (var-binding->mutations formal context)))))
    (make-simple-expl "for identifying lambdas with non-mutated formals")))
```

The raw predicate above assumes that `lambda-exps` is a list of `lambda` expressions. In all cases, raw predicates must know which viewcode expressions are of interest in the `vc-exps` list part of the characteristic vform interface (this list is arranged by the predicate's invoker). The raw predicate also assumes that the alpha view has been computed on `context`. It can therefore use the alpha view retrieval function `var-binding->mutations` to find mutations, with respect to `context`. If `lambda-exps` were to be the list of `dispatch-expression`'s `lambda` expression and `context` were to be the context containing `dispatch-expression`, this predicate would return true, indicating that `dispatch-expression`'s formal parameter `exp` is not mutated in `dispatch-expression`'s body.

Another example is the following predicate, which returns true if every viewcode expression given in `vc-exps` is a `lambda` expression of one formal. The predicate uses the function `type/lambda?` from the default viewcode view. `type/lambda?` returns true if its argument is a viewcode `lambda` expression.

```
(define predicate/lambda-one-formal
  (make-predicate (lambda (vform context vc-exps)
                    (and (for-all? vc-exps type/lambda?)
                         (for-all? (map lambda/formals vc-exps)
                           (lambda (formals) (= (length formals) 1)))))
    (make-simple-expl "for identifying lambda expressions with one formal")))
```

4.4.3 Actions

An action unconditionally modifies viewcode expressions. Some common uses for actions include replacing operators, changing representations, inlining code, and moving code.

`(make-action modifier explanation)`

Returns a primitive vform. Both the modifier and explanation must have the characteristic vform interface. When invoked, the modifier makes alterations to viewcode expressions and returns a non false value if successful. The explanation produces a description of the modifier.

Like predicates, actions can use views. Unlike predicates, actions tend to use views to find expressions; that is, to navigate through the viewcode. An action's implementation typically decides which expressions are needed to make the desired transformation, finds those expressions using the appropriate views, then replaces, deletes, or inserts into the viewcode to produce the results. For instance, suppose we wanted to implement an action that took a list of `lambda` expressions, and changed the name of (i.e., alpha renames) each formal and its references to `foo-bar`. To do this, we need to gather each formal's variable references. This can be accomplished using the alpha view function `variable-binding->var-refs`. Once we have gathered the formals, we can use the viewcode function `replace` (given earlier) to directly modify the viewcode. The code below implements this action.

```
(define action/rename-foo-bar
   (make-action (lambda (vform context vc-exps)
                  (for-each (lambda (formal)
                              (for-each (lambda (var) (replace var 'foo-bar))
                                        (var-binding->var-refs formal context))
                              (replace formal 'foo-bar))
                            (append-map lambda/formals vc-exps))
                  #t)
     (make-simple-expl "for renaming var references and their lambda formals to foo-bar")))
```

4.4.4 Dispatchers

Dispatchers are functionally similar to what are commonly referred to as conditional rewrite rules (see Figure 3-3). They differ from typical rewrite rules in that dispatchers can depend on non-local program information (via views and predicates), can be non semantics preserving (via actions), and as discussed in Section 4.10, can interact with the user. This makes dispatchers more expressive than the rewrite rules found in other program transformation systems I surveyed[78].

A dispatcher is constructed as a combination of a predicate and an action. A dispatcher first invokes the predicate. If the predicate returns a true value, the action is invoked. The dispatcher returns the action's return value, or `#f` if the dispatcher was not applicable.

`(make-dispatcher predicate-vform action-vform expl-combiner)`

Returns a dispatcher whose predicate is `predicate-vform` and whose action is `action-vform`. `expl-combiner` combines the predicate's and action's respective explanations.

Dispatchers are typically implemented once a set of predicates and corresponding actions have been specified or written. In order to write a dispatcher, a programmer must determine which predicate validates an action. For example, suppose that we wanted to rename all the non-mutated formals in a set of `lambda` expressions to `foo-bar`. We could do this by combining `predicate/no-mutations` with `action/rename-foo-bar` (both defined above) into a dispatcher. The following code defines such a dispatcher (the explanation combiner is discussed later, in Section 4.7). If this dispatcher were passed `dispatch-expression`'s `lambda` expression and context, it would rename every occurrence of the variable `exp` to `foo-bar` in `dispatch-expression`.

```
(define dispatcher/rename-to-foo-bar
   (make-dispatcher predicate/no-mutations action/rename-foo-bar *default-expl-combiner*))
```

4.4.5 Rules and Couplers

A dispatcher is a kind of rule. A rule is some combination of predicates, actions, dispatchers, or other rules (referred to as sub-vforms in the interface description below). A rule controls how its sub-vforms are applied via its control structure. A control structure accepts a list of vforms (i.e., the `sub-vforms` below) and returns a procedure with the characteristic vform interface.

`(combine-vforms control-structure expl-combiner sub-vforms)`

Returns a rule with the given control structure, explanation combiner, and sub-vforms. Internally, applies `control-structure` and `expl-combiner` to `sub-vforms`

Simple control structures just invoke the sub-vforms and combine the results. ViewForm provides various kinds of simple control structures (e.g., combine results conjunctively or disjunctively). Users can also write their own control structures. For example, suppose we want a control structure that invokes each sub-vform (via the `do-vform` function, described next in Section 4.5) sequentially, on the given context and viewcode expression list. This control structure would return a true value if all sub-vforms returned true values, but would immediately stop and return a false value upon encountering a sub-vform that returned a false value. This kind of control structure would be useful when combining a set of predicates that must all be true. The control structure below implements this specification:

```
(define *conjunctive-control-structure*
  (lambda (sub-vforms)
    (lambda (vform context vc-exps)
      (for-all? sub-vforms
        (lambda (sub-vform) (do-vform sub-vform context vc-exps))))))
```

We can use this control structure to make a compound predicate; a rule made up of a combination of two or more predicates. Suppose we wanted a compound predicate that tested whether a set of viewcode expressions were all `lambda` expressions of one non-mutated formal. This compound predicate is a combination of the two predicates given in Section 4.4.2. It can be created as follows:

```
(define predicate/lambda-non-mutated-formal
  (combine-vforms *conjunctive-control-structure*
                  *default-expl-combiner*
                  (list predicate/lambda-one-formal predicate/no-mutations)))
```

A dispatcher using this predicate can also be defined:

```
(define dispatcher/rename-non-mutated-to-foo-bar
   (make-dispatcher predicate/lambda-non-mutated-formal
                    action/rename-foo-bar
                    *default-expl-combiner*))
```

4.4.5.1 Couplers

A coupler is a kind of rule that carries out an entire coupling. A coupler's duties correspond to the coupler production step (i.e., Step v) of the view-based abstraction methodology. These duties are: creating the appropriate contexts, ensuring the proper views are computed on the contexts, creating viewcode expression lists, invoking sub-vforms (e.g., dispatchers), and registering a successful coupling with the view invalidation/recoupler. An implementation of these duties is demonstrated below, in the creation of a sample control structure and coupler.

Suppose we wanted to write a coupler that renamed all non-mutated variables in `lambda` expressions with one formal to `foo-bar`. We begin by writing a control structure that will be used to construct our coupler. Like all control structures, this one assumes that the vforms passed in to it will attempt to carry out the desired coupling. We specify that the control structure will be passed a filename containing the expressions to be modified. The control structure can thus use this filename to create a context using `make-context-from-files`. The next step is to ensure that the proper views have been invoked. Since we will be using the compound predicate defined above, our control structure must ensure that the alpha view is computed on the context. Next, the control structure must create a candidate list of viewcode expressions to be given to the dispatcher. We want this list to include all top-level `lambda` expressions in the context. To create this list of candidates, the control structure first filters out any context expressions that are not top-level `define`'s, then extracts the `define` expression values from those that remain. The control

structure is now ready to invoke the passed-in vforms. It invokes each vform, then checks if any one of them succeeded. The last step is to register a successful coupling with the view invalidation/recoupler ("success" means that at least one vform returned a non-false value).

```
(define (make-foo-bar-control-structure file1)
  (lambda (vforms)
    (lambda (vform context vc-exps)
      (let* ((context (make-context-from-files file1))
             (maybe-lambdas (map define-var/value
                                 (list-transform-positive (context/exps context)
                                   type/define-var?))))
        (ensure-alpha-view-computed! context)
        (if (list-search-positive (map (lambda (specific-vform)
                                         (do-vform specific-vform context maybe-lambdas))
                                       vforms)
              identity-function)
            (begin (register-coupling! context vform) context)
            #f)))))
```

This control structure can now be used to define our desired rule (the implementation of the higher-level explanation combiner `verbose-expl-combiner` is given in Appendix B):

```
(define coupler/rename-to-foo-bar
   (combine-vforms (make-foo-bar-control-structure "test-file.scm")
                   (verbose-expl-combiner "renaming non-mutated lambda formals to foo-bar")
                   (list dispatcher/rename-non-mutated-to-foo-bar)))
```

4.5 Vforms

ViewForm builds its coupling constructs and views from *vforms*. Vforms are first-class objects that can be composed into higher-level vforms. A vform consists of an identifier, a control structure, an explanation combiner, and a set of sub-vforms. Non-primitive vforms are constructed using `make-vform`, given below.

> **`(make-vform identifier control-structure expl-creator sub-vforms)`**
>
> Returns a vform. `identifier` is a symbol. `expl-creator` and `control-structure` take a list of vforms (i.e., `sub-vforms`) and return a procedure with the characteristic vform interface.

The control structure's job is to manage the invocation of the sub-vforms, while the explanation creator's job is to combine the sub-vform explanations into a single explanation. While `make-vform` is not used in other parts of the dissertation, it illustrates and substantiates the deep commonality between coupling constructs, as suggested by the similarity of their formalization in Section 3.7.3. This commonality is typified by the *characteristic vform interface*:

> **`vform: : delegating-vform x context x vc-exps -> any-Scheme-value`**

This is the signature for every kind of coupling construct implemented in ViewForm. So far, the purpose of `context` and `vc-exps` has been demonstrated, but the purpose of `delegating-vform` has

not. Earlier in this dissertation, the coupling constructs were described to be delegation based. This is what `delegating-vform` is for. `delegating-vform` is the vform that should be invoked for a delegated (i.e., recursive) vform invocation. Before showing how this is done, the remaining interface functions for vforms are given. Note that a vform's sub-vforms can be inserted, replaced, modified, or examined. This feature was designed with the benefits of translucent procedures[80] in mind.

`(vform/vforms vform)`
`(set-vform/vforms! vform)`

Retrieves or sets the sub-vforms in `vform`.

Invoking a vform is done using either of the following two forms:

`(do-vform vform context vc-exps)`

Returns the result of invoking `vform` on the viewcode expression list `vc-exps`, with respect to `context`.

`(delegate-vform vform context vc-exps delegating-vform)`

Returns the result of invoking `vform` on the list of viewcode expressions `vc-exps` on behalf of `delegating-vform`, with respect to `context`.

The difference between the two ways of invoking vforms is that `do-vform` will invoke `vform` on its behalf whereas `delegate-vform` will invoke `vform` on `delegating-vform`'s behalf. This is straightforward delegation, similar to what is found in [1].‡‡

For example, referring to `coupler/rename-to-foo-bar`'s control structure above, `do-vform` is used to invoke each `specific-vform` (which in this case, is just `dispatcher/rename-non-mutated-to-foo-bar`) on behalf of `specific-vform`. This means that `dispatcher/rename-non-mutated-to-foo-bar` will be given a reference to itself as its delegating vform (as opposed to a reference to `coupler/rename-to-foo-bar`). Dispatchers, however, always use `delegate-vform` when invoking their predicate and action. This means that both the dispatcher's predicate, `predicate/lambda-non-mutated-formal`, and its action, `action/rename-foo-bar`, will receive a reference to `dispatcher/rename-non-mutated-to-foo-bar` as their delegating vforms (as opposed to references to themselves). This implies that the expression `(do-vform vform context …)` evaluated within the body of the predicate or action *would invoke the dispatcher.*

This can be contrasted with what would happen if `make-foo-bar-control-structure` had used `(delegate-vform specific-vform context maybe-lambdas vform)` instead of `(do-vform specific-vform context maybe-lambdas)`. In this case, the dispatcher would receive a reference to `coupler/rename-`

‡‡ In implementation terms, `do-vform` passes `vform` as the first argument to the vform actually run, whereas `delegate-vform` passes `delegating-vform` as the first argument to the vform actually run.

`to-foo-bar` as it delegating vform (assuming the coupler was invoked using `do-vform`). The dispatcher would delegate this down to its predicate and action. If the predicate or action were now to evaluate the expression `(do-vform vform context …)`, *the coupler would be invoked* (as opposed to the dispatcher being invoked).

Delegation provides a variety of benefits. One important benefit is the ability to reuse vforms. For instance, if we layer a new vform over an existing vform which uses `delegate-vform` internally, we can assume that any recursive vform invocations from within the existing vform will be to our new vform (if the existing vform had used `do-vform`, a recursive invocation would have been to the existing vform itself). This is no more than delegation's version of class-based inheritance in an object-oriented language. This technique was used to implement MIT Scheme macros over existing views, by simply layering new vforms over the existing views. This technique is demonstrated in Section 4.6.4.

Since `make-vform` takes vforms as arguments, there must be a way to create truly primitive vforms. ViewForm supports the creation of primitive vforms with the following functions. These functions were used to define constructors such as `make-predicate`, `make-action`, and `make-view`.

`(make-primitive-vform identifier vform-proc explanation-proc)`

Returns a primitive vform. `vform-proc` and `explanation-proc` are both procedures having the characteristic vform interface

`(designate-primitive-vform-identifier! identifier)`

Designates `identifier` as a primitive vform identifier

4.6 Views

Under the view-based abstraction model, views provide a mapping between context expressions and their properties with respect to a context. ViewForm provides several default views: the viewcode view, the alpha view, and the liar view. ViewForm also provides a higher-level view, the walker view. The walker view is a code walker that can be used as the basis for creating other views.

Views can be built as combinations of vforms. For example, a view can be implemented as a rule with a preemptive sequential control structure (like `*conjunctive-control-structure*`). This rule can invoke a set of dispatchers on each given viewcode expression. The predicates within the dispatchers can test for the presence of a particular type of viewcode expression (e.g., using `type/lambda?`, `type/define-var?`, etc.). Instead of modifying the viewcode, however, the corresponding actions can modify the view's information data structure. The default ViewForm views

(except for the viewcode view) are all implemented this way.[§§] The corresponding function for creating a view is:

(make-view control-structure expl-combiner vforms)

Returns a vform that computes view information. `control-structure` and `expl-combiner` both take a list of vforms as their parameter, and return a procedure with the characteristic vform interface. `vforms` is the list of vforms passed to `control-structure` and `expl-combiner`.

Unlike predicates, which normally capture a programmer's criteria for validating a particular viewcode modification, views capture the program properties necessary to actually validate those criteria. This separation between computing specific coupling preconditions and computing general program properties simplifies a predicate's construction for two reasons. The first is by abstracting away the usually complicated program analysis implementation details. That is, using views is simpler than writing them. The second reason is that views make selected properties salient, while hiding other properties. For example, a naming view does not also return (or compute) the results of a program's control flow or data flow.

Each view provides an interface function for computing, retrieving, merging, and copying its program property mapping. The interfaces for ViewForm's default views are discussed later in this section. I considered developing a canonical interface for accessing view information, but concluded that such a standard would only constrain the kinds of program information expressible by views. What might be less constraining would be to standardize an interface for particular kinds of information, such as data-flow information. This would facilitate reuse or substitution of views that capture the same information to differing precisions or using differing amounts of computational resources.

Programmers are free to develop their own views. Besides the absence of a view for computing a set of desired program properties, various concerns can motivate a programmer to write new views. Two such concerns are precision and performance, especially for views that collect non-local information such as data-flow quantities. These analyses tend to tradeoff precision for time, or time for space[3]. Depending on the size of the viewed context or the desired precision, a programmer may want a view that makes different tradeoffs. Since computing views can be the performance bottleneck in ViewForm, carefully selecting rules that use views with the proper performance characteristics might be required to successfully complete a coupling. A simple example of creating a view is given later, in Section 4.6.4. A more complex example is provided in Chapter 5. For now, the remaining functions used to create and manage views are:

[§§] The viewcode view differs only because it actually implements the `type/<exp>?` functions.

(register-view! view merge merge! copy init)

Registers a view, along with a function to merge views, a function to destructively merge views, a function to copy views, and an initializing value (the merging and copying functions are called by `merge-contexts` and `merge-contexts!`).

(view/merge-proc view) (view/merge!-proc view) (view/copy-proc view) (view/init view)

Returns the respective procedures and the initialization value registered by `register-view!`

(ensure-<view-name>-view-computed! context #!optional vc-exps)

A user-defined function that ensures the named view is computed on `vc-exps` with respect to `context`. If the list `vc-exps` is not given, assumes that the named view is to be computed on all expressions in `context`.

Since view information can certainly be constructed by hand, views can be regarded as machine-readable descriptions of program properties. Views can thus serve as specification languages for properties that may otherwise have been English language descriptions of a program's implementation. I do not explore this use for views, but note that specification languages is an active field of research.

The default views provided by ViewForm are described below. The descriptions contain the more interesting view interface functions.

4.6.1 Viewcode View

The viewcode view is available by default on all viewcode expressions. It provides enhanced viewcode navigation functionality, allowing movement both up and down the expression tree. It also provides fine-grained expression typing. The viewcode view is automatically computed on any Scheme expressions parsed by ViewForm.

The viewcode view provides the following functions for each viewcode expression:

(exp/type exp)

Returns the expression's type.

(type/<type>? exp)

Returns `#t` if `exp` is of type `<type>`.

(up-link-n exp n)

Traverses `n` indirections up the expression tree, and returns the resulting expression. Up-linking a top-level context expression returns the value of the top-level variable `*top-level*`.

When expressions are parsed, the viewcode view computes an expression type for each viewcode expression and recursively up links each expression. The viewcode view provides predicate functions to discriminate among these expression types. The functions `type/lambda?` and

`type/define-var?` used above, for example, return true if the viewcode expression passed to them are of the form `(lambda (…) …)` or `(define <var> …)`, respectively. The full set of expression-type discriminator functions are given in Appendix A.

Up linking provides real viewcode navigability. Compound Scheme expressions are represented as lists in viewcode, but lists are not doubly linked by default in Scheme. Up linking provides the back link that would otherwise be present in a doubly-linked list. Without this functionality, examining or depending on parent expressions would not be expressible. The following transcript demonstrates up linking assuming that `disp-exp` is the viewcode for `(define dispatch-expression (lambda (exp) …)` in Figure 1-2:

```
(define (up-link exp) (up-link-n exp 1))
(eq? disp-exp (up-link (car disp-exp))) ➔ #t     ;up-linking is the inverse of car and cdr
(first (lambda/formals (define-var/value disp-exp))) ➔ exp
(up-link (first (lambda/formals (define-var/value disp-exp)))) ➔ (exp)
(up-link (lambda/formals (define-var/value disp-exp))) ➔ (lambda (exp) (cond …))
(up-link (define-var/value disp-exp)) ➔ (dispatch-expression (lambda (exp) (cond …)))
(eq? (up-link disp-exp) *top-level*) ➔ #t
```

As discussed earlier in Section 4.2, some Scheme expressions, such as numbers and symbols, are not mutable. This makes it difficult to up link them via their identity. For this reason, viewcode represents these expressions using structures that do have identity, the *enclosures* whose interface was given in Section 4.2. Enclosures are also typed by the viewcode view with functions such as `type/symbol?`.

4.6.2 Alpha View

The alpha view performs the equivalent of an alpha conversion. Unlike most alpha conversions, however, the alpha view does not modify variable names. Instead, it maintains a two-way association between each variable reference (and mutation) and its corresponding variable binding site. A variable binding site is an enclosure holding a `lambda` formal parameter, a `define` variable, or any other identifier that equivalently introduces a new variable.

The alpha view functions are:

`(ensure-alpha-view-computed! context)`

Ensures that the alpha view has been computed on the expressions in `context`

`(context/alpha context)  (set-context/alpha! context view-mapping)`

Returns or sets the view mapping associated with `context`, the former returns `#f` if none exists

`(var-ref->var-binding var-ref)`

`var-ref` is a variable reference. Returns a list of `var-ref`'s variable bindings sites

(var-binding->var-refs var-binding)

`var-binding` is a variable binding. Returns a list of variable references to `var-binding`

(var-binding->mutations var-binding context)

Returns a list of variables in `set!` and `fluid-let` expressions (and `define` expressions, if redefined) that refer to the variable in the binding site `var-binding` with respect to `context`

(var-mutation->binding var-mutation context)

Given a mutation variable `var-mutation`, returns the corresponding variable binding site or `#f` if none exists

(top-level-defined? var-name context)

`var-name` is a symbol. Returns the top-level variable binding site in the context for the variable named `var-name`. Returns `#f` if none exists

(top-level-names context)

Returns a list of all the binding sites of each top-level name defined in `context`

(unbound-var-lookups-in-context context)

Returns a list of all the unbound variable references in `context`.

(unbound-mutations-in-context   context)

Returns a list of all the unbound variable mutations in `context`

(merge-alpha-views a1 a2)

Returns a new alpha view containing the merged alpha views `a1` and `a2`

(merge-alpha! a1 a2)

Destructively merges alpha view `a2` into `a1`

4.6.3 Liar View

The liar view is a data-flow analysis based on LIAR[79]. It computes a graph whose nodes are viewcode expressions and whose edges denote that a run-time value can flow from one node's corresponding expression to another. Some compilers perform similar kinds of graphs for use in optimizing programs, although the graphs are typically computed intraprocedurally and not interprocedurally.

More specifically, the liar view collects the following information on each viewcode expression:

- A set of *producers*; expressions that produce values that can potentially end up being the given expression's return value.
- A set of *consumers*; expressions that can potentially receive or consume any of the expression's return values.

In Figure 3-2, for example, the binding variable `f`'s producers are the variables `proc` and `cdr`, and the `lambda` expression. Its consumer is the variable reference `f`. An expression's producers are classified into the following categories:

- Predefined procedures that can potentially be the value of the given expression
- User-defined procedures that can potentially be the value of the given expression
- Special forms that can potentially produce the value of the given expression
- Constants that can potentially be the value of the given expression

The relevant liar interface functions are:

`(ensure-liar-view-computed! context)`

Ensures that the liar view has been computed on the expressions in `context`

`(context/liar context)  (set-context/liar! context view)`

Returns or sets the liar data-flow analysis view associated with `context`. Returns `#f` if none exists

`(exp->recvs exp context)`

Returns a list of `exp`'s consumers, with respect to `context`

`(exp->prods exp context)`

Returns a list of `exp`'s producers, with respect to `context`

```
(exp->primops  exp context)
(exp->procs    exp context)
(exp->sp-forms exp context)
(exp->consts   exp context)
```

Returns a sub-list of `exp`'s producers whose returns values are produced primitively, are procedures, are produced by special forms, or are constants, respectively

`(merge-liar-views! l1 l2)`

Destructively merges liar view `l2` into `l1`

`(merge-liar-views l1 l2)`

Returns a new liar view that contains the merged information from liar views `l1` and `l2`

The liar view also provides a function for determining the type of an expression's return value. These types are different from the viewcode view expression types, which correspond to the syntactic (and hence, static) grammatical property of an expression. For example, the viewcode expression `(fix:1+ 1 2)` satisfies `type/combination?`, whereas its return value is of type `*non-negative-fix*`. These dynamic return value types are listed in Appendix A, and are computed by the following function:

(exp->return-types exp context)

Returns a list of exp's possible return types (see Appendix A.2), with respect to context.

One of the liar view's important properties is its performance. This property is important to programmers choosing between the liar view and other data-flow views. The liar view invocation was optimized to trade space for time. This means that the liar view may not be able to compute its information on large contexts. To counter balance this tradeoff, the liar merging operations were optimized to trade time for space. This allows the liar view to be computed on large contexts by merging several smaller contexts on which the liar view has been invoked.

4.6.4 Higher-Order Views

ViewForm provides a higher-order view, called the walker view. This view is a top-down code walker. By default, it recursively *delegates* itself to each sub-expression it finds. It is a higher order view because it generalizes the notion of a code walker, and because it needs to be "customized" to do specific work. This customization happens by combining the walker vform with dispatchers (or other rules) that take action on specific kinds of expressions. In essence, these new dispatchers override the default walker's dispatchers for those expressions. In order to continue walking down the viewcode tree after such an override, however, the walker's views still need to be invoked (in CLOS parlance, this means calling `call-next-method`). To do this, the dispatcher doing the work must be carefully combined with the walker vform. Selecting a control structure that invokes the new dispatchers first, then unconditionally invokes the walker vform is sufficient. This is illustrated in an example below.

walker-vform

A compound rule that recursively delegates down all the viewcode expressions in the context it is passed. Can be combined with other rules to produce viewcode walkers

Suppose we want to define a view that maintains a list of all `lambda` expressions that have one non-mutated formal, within a given set of viewcode expressions (such as `dispatch-expression`'s `lambda`). To implement this "lambda walker view", we go through a series of five steps:

1. Implement the merging and copying procedures, as well as the view initialization value
2. Implement the view dispatchers
3. Combine the dispatchers into a view
4. Register the view
5. Provide a way to invoke the view

For our desired view, Step 1 involves deciding upon a representation for the view information. For this example, let us choose a simple list. The Step 1 code is therefore implemented as list operations:

```
(define (lambda-walker-merge lambda-view1 lambda-view2)
  (append  lambda-view1 (list-copy lambda-view2)))
(define (lambda-walker-merge! lambda-view1 lambda-view2)
  (append! lambda-view1 (list-copy lambda-view2)))
(define lambda-walker-copy list-copy)
(define lambda-walker-init '())
```

For Step 2, we have previously defined the predicate for our desired view: `predicate/lambda-non-mutated-formal`. We need only implement an action that stores a `lambda` expression. We write the action to assume that the first expression in the viewcode list is the `lambda` expression to be stored into the appropriate lambda walker information list. This will be the list retrieved and set by `retrieve-view` and `set-view!` (described earlier). The action is implemented below, followed by the dispatcher combining the predicate and action.

```
(define action/store-lambda
  (make-action
    (lambda (vform context vc-exps)
     (set-view! *lambda-view* context
      (cons (car vc-exps) (or (retrieve-view *lambda-view* context) lambda-walker-init)))
     #t)
   (make-simple-expl "for storing a combination")))

(define walker/lambda-finder
  (make-dispatcher predicate/lambda-non-mutated-formal
                   action/store-lambda
                   (verbose-expl-combiner "for collecting combinations")))
```

For Step 3, we create the view itself, using `make-view`:

```
(define *lambda-view*
  (make-view (lambda (vforms)
               (let ((combined-vform (apply combine-vforms-do-all vforms)))
                 (lambda (vform context vc-exps)
                   (set-view! *lambda-view* context lambda-walker-init)
                   (ensure-alpha-view-computed! context)
                   (for-each (lambda (vc-exp) (do-vform combined-vform context (list vc-exp)))
                             vc-exps)
                   #t)))
             (verbose-expl-combiner "lambda-walker combination collector")
             (list walker/lambda-finder *walker-vform*)))
```

The view's control structure begins by creating `combined-vform`, a combination of our dispatcher, `walker/lambda-finder`, with `*walker-vform*` using `combine-vforms-do-all`. The method of combination is important. We want a combined vform that always invokes `walker/lambda-finder` followed by `*walker-vform*`, the former collecting our `lambda` expression and the latter traversing the viewcode expression tree. `combine-vforms-do-all` does this, as its name suggests.

The control structure next clears the lambda walker view for the given context, and ensures the alpha view has been computed for that context. Once this is done, the viewcode expressions are traversed, one by one. Notice that in the call to `do-vform`, the vform invoked is `combined-`

vform. Since *walker-vform* delegates to whatever vform invoked it (according to its specification), the delegation will end up invoking combined-vform on every sub-expression encountered by *walker-vform*. This is exactly the behavior we want, combined-vform ensures that walker/lambda-finder is always invoked.

Step 4 is straightforward, using the functions defined in Step 1:

```
(register-view! *lambda-view* lambda-walker-merge
                              lambda-walker-merge!
                              lambda-walker-copy
                              lambda-walker-init)
```

Step 5 is also straightforward. ensure-lambda-view-computed! invokes the *lambda-view* vform. After invoking the view, retrieve-view can be used to retrieve the list of lambda expressions associated with a context.

```
(define (ensure-lambda-view-computed! context #!optional vc-exps)
  (if (not (retrieve-view *lambda-view* context))
      (if (default-object? vc-exps)
          (do-vform *lambda-view* context (context/exps context))
          (do-vform *lambda-view* context vc-exps))))
```

While this example implements a relatively simple view, it illustrates two important, although somewhat polar points. The first is that views are more complicated to write than other kinds of vforms. The second point, on the other hand, is that given a code template such as the one above, it is easy to substitute a dispatcher for walker/lambda-finder and rename a few view functions to quickly and easily create new kinds of views. This reduces the problem of creating view akin to the lambda walker view to the problem of writing the right dispatcher. While not all views can be written this way, many simple ones can.

4.7 Explanations

An *explanation* is not a coupling construct. Rather, explanations are associated with coupling constructs. An explanation produces a description of what its associated coupling construct vform does. This description provides the user with an effective way of determining what the vform does, either directly or from within a control structure (e.g., when a predicate returns false, the offending predicate can be explained). The description can be in a natural, human language or it can be in a machine-readable language. I use the former method, but the latter method could be useful, for example, as a machine-checkable condition for verifying a coupling's modifications. Explanations are created during the creation of predicates and actions and during the combination of vforms (as described earlier). When combining explanations, more abstract explanations can be provided to give more concise descriptions of the corresponding, combined coupling constructs.

The explanations used in this dissertation are created via the following function:

(make-simple-expl text)

`text` is a string. Returns an explanation that produces, to the standard output, a carriage return and "<type> text" where <type> is the explained vform's identifier

The following function invokes an explanation:

(explain! vform context vc-exps)

Produces an explanation of `vform`'s functionality on the viewcode expressions `vc-exps` with respect to the context, `context`.

The following code demonstrates how an explanation combiner is implemented. This combiner produces an explanation that, when explained, identifies the type of the vform being explained and then recursively explains each of the combined vforms:

```
(define *default-expl-combiner*
 (lambda (vforms)
   (lambda (vform context vc-exps)
     (format #t "~S containing " (vform/identifier vform))
     (for-each (lambda (vform) (explain! vform context vc-exps))
               vforms))))
```

Various functions for embedding explanations into predicates and actions are also provided by ViewForm:

(make-explained-predicate raw-predicate text-string)
(make-explained-action modifier text-string)

Return vforms similar to those returned by `make-predicate` and `make-action`, except that, when explained, the vforms return a string of the form, "predicate for identifying <text-string>" and "action for <text-string>", respectively

4.8 View Invalidation/Recoupler

The ViewForm view invalidation/recoupler maintains the information needed to invalidate and recompute couplings. When invoked, the view invalidation/recoupler checks through its registry and determines which coupled contexts have been invalidated. If the context originated from a file, for example, it checks that file's most recent modification date. When it finds an invalid coupling, the view invalidation/recoupler performs a recoupling by re-invoking the coupler that originally coupled the context. The view invalidation/recoupler interface functions are:

(register-coupling! vform context)

Registers a coupling rendered by `vform` on `context`

(unregister-coupling! vform context)

Unregisters a previously registered coupling

(vir!)

Recomputes any invalidated couplings

4.9 Dispatcher Example

Chapter 3 presented the view-based abstraction methodology. ViewForm was implemented to allow programmers to experiment with implementation coupling using the view-based abstraction methodology. This section works through the methodology on the dispatch example first presented in Chapter 1. The goal is to couple the `dispatch-expression` code in Figure 1-2 with `type?`'s implementation details from the code in Figure 1-3 to produce the coupled (and view-invalidation/recoupler registered) `dispatch-expression` code in Figure 1-4. This example relies heavily on previously discussed aspects of the dispatch example, and on previously presented sample code.

4.9.1 Boundary/Context Identification

The goal for this step is to decide how to divide the program expressions into contexts. As previously explained, the dispatch example has two clear contexts: the uncoupled code for `dispatch-example` and the `process-<exp>` procedures from Figure 1-2, and `type?`, `exp-type`, and the code for computing expression types in Figure 1-3. This code is stored in the files "linear-cond.scm" and "types.scm", respectively. Later, these files will be passed to our coupler's control structure which will create contexts from them. No code need be written at this point.

4.9.2 Precondition/Predicate Development

The goal for this step is to develop a set of preconditions for the coupling, then implement them as predicates. Since we already have two adequate preconditions P_1' and P_2' from Section 3.2.2, we choose those. To implement these preconditions, we will write five predicates and then combine them conjunctively. We begin by implementing the simplest precondition, which is that predefined primitives are not re-defined. This can be accomplished by checking for the MIT Scheme declaration `(declare (usual-integrations))`. This declaration allows the compiler to open code primitives, in essence, nullifying any re-definitions. The corresponding predicate's implementation is:

```
(define predicate/usual-integrations (make-predicate/find-declaration 'usual-integrations))
```

The ViewForm library function `make-predicate/find-declaration` (whose definition is given in Appendix B) looks for the given declaration in its context's viewcode expressions. The next precondition we implement is the one stating that `dispatch-expression`'s formal (i.e., `exp`), is not mutated. To make this predicate more aesthetic, we assume the existence of the function `vc-`

exps/dispatch-exp-value. This function will select the viewcode expression for dispatch-expression's value (which we expect to be a lambda expression) out of the predicate's viewcode expression list. We will define this function later, when combining the predicates. The resulting predicate, predicate/no-mutations, uses the previously specified alpha view function var-binding->mutations. When working on the code analysis/view development step, we must remember that this predicate requires that the alpha view be computed.

```
(define predicate/no-mutations
  (make-explained-predicate
    (lambda (vform context vc-exps)
      (null? (var-binding->mutations
               (first (lambda/formals (vc-exps/dispatch-exp-value vc-exps))) context)))
    "identifying variable bindings with no mutations"))
```

The next precondition we implement checks the cond clauses in dispatch-expression. For this precondition, we defined two predicates; one for a cond clause's predicate, and one for a cond clause's consequents. We will then combine the predicates into a compound predicate. For the cond clause predicate, we assume, for aesthetics, the existence of a function vc-exps/dispatch-exp-formals that selects the viewcode for dispatch-expression's lambda's formals from vc-exps. We will define this function later. The predicate below ensures that the cond predicate clause is a combination that only calls type? on a quoted symbol and on dispatch-expression's first formal. It uses the viewcode function type/combination?, and the ViewForm library functions only-calls?, quoted-symbol?, and var-bound-by?. For the interested reader, these functions are defined in terms of basic ViewForm functions in Appendix B.

```
(define predicate/clause-predicate
  (make-explained-predicate
   (lambda (vform context vc-exps)
     (let ((predicate  (clause/predicate (vc-exps/dispatch-exp-value   vc-exps)))
           (exp-formal (first            (vc-exps/dispatch-exp-formals vc-exps))))
       (and (type/combination? predicate)
            (only-calls? 'type? predicate context)
            (quoted-symbol? (second predicate))
            (var-bound-by?  (third  predicate) exp-formal context))))
   "validating clause predicates for the dispatch example"))
```

The precondition for the cond clause consequents ensures that their return value is a combination whose only argument is dispatch-expression's lambda's formal (expected to be exp).

```
(define predicate/clause-consequent
  (make-explained-predicate
    (lambda (vform context vc-exps)
     (let ((consequent (last  (clause/consequents (vc-exp/dispatch-exp-value vc-exps))))
           (exp-formal (first (vc-exp/dispatch-exp-formals vc-exps))))
       (and (type/combination? consequent)
            (= (length consequent) 2)
            (var-bound-by?  (second consequent) exp-formal context))))
   "validating clause consequents for the dispatch example"))
```

Next, we combine the above two predicates into one, compound predicate. In addition to invoking the above two predicates on all of the `cond` clauses, the compound predicate also ensures that `dispatch-expression`'s value is a `lambda` expression whose body is a `cond` (thus testing another part of P_1'). This predicate, therefore, completes the conditions given in P_1'.

```
(define predicate/dispatch-expression
  (combine-vforms
   (lambda (vforms)
     (lambda (vform context vc-exps)
       (let ((dispatch-exp-value (vc-exps/dispatch-val vc-exps)))
         (and (type/lambda? dispatch-exp-value)
              (type/cond? (lambda/body dispatch-exp-value))
              (for-all? (cond/clauses (lambda/body dispatch-exp-value))
                (lambda (clause)
                  (or (type/else? (clause/predicate clause))
                      (for-all? vforms
                        (lambda (vform)
                          (do-vform vform context
                                    (list clause (lambda/formals dispatch-exp-value))))))))))))
   *id-expl-combiner*
   (list predicate/clause-predicate predicate/clause-consequent)))

(define vc-exps/dispatch-exp-value   car)
(define vc-exps/dispatch-exp-formals cadr)
```

Since this compound predicate creates the viewcode expression list for the two predicates above, we can define our selectors (i.e., `vc-exps/dispatch-exp-value` and `vc-exps/dispatch-exp-formals`) according to how we build the viewcode expression list. Notice, however, that the compound predicate relies upon `vc-exps/dispatch-val`, another viewcode expression list selector. We define this later, when writing the coupler.

The final precondition to implement is P_2'. The predicate below implements this precondition by navigating into `type?`'s body and ensuring it is a combination that calls `eq?` on `type?`'s formal parameter. Since `eq?` is symmetric, we must check both `eq?`'s first and second argument.

```
(define predicate/type?
  (make-explained-predicate
   (lambda (vform context vc-exps)
     (let* ((type?-lambda     (vc-exps/type?-lambda vc-exps))
            (type?-exp-formal (first (lambda/formals type?-lambda)))
            (type?-body       (lambda/body type?-lambda)))
       (and (type/combination? type?-body)
            (only-calls? 'eq? type?-body context)
            (or (var-bound-by? (second type?-body) type?-exp-formal context)
                (var-bound-by? (third  type?-body) type?-exp-formal context)))))
   "type?'s body eq?'ing its formal to a type symbol expression"))
```

4.9.3 Code Analysis/View Development

The goal of this step is to determine which views are being used and to ensure that those views exist. The predicates above need two views: a naming view and a data-flow view (used by

only-calls?). ViewForm provides two such views: the alpha view and the liar view, respectively. Since there are no views to write, this step simply involves noting that the coupler should invoke these views on the contexts created in the boundary/context identification step .

4.9.4 Modification/Action Development

Our goal in this step is to write an action that expresses a transformation from the code in Figure 1-2 to the code in Figure 1-4. To do this best, we must minimize the scope of the modification. That is, we must express a transformation that depends as little as possible on surrounding code that a programmer performing a manual coupling would not need to examine. The code in Figure 1-4 depends on the uncoupled dispatch-expression code in three ways: the type symbols, the process-<exp> expressions, and the else clause consequents. Since our predicates have constrained the syntactic and semantic structure of the dispatch-expression code, we can collect these expressions directly from the cond clauses:

```
(define collect-type-symbols
  (make-cond-clause-collector type/else?
                              (lambda (cond-clause)
                                (second (clause/predicate cond-clause)))))

(define collect-process-procs
  (make-cond-clause-collector type/else?
                              (lambda (cond-clause)
                                (combination/proc (last (clause/consequents cond-clause))))))

(define collect-else-consequents
  (let ((collector (make-cond-clause-collector (lambda (predicate-clause)
                                                 (not (type/else? predicate-clause)))
                                               (lambda (cond-clause)
                                                 (clause/consequents cond-clause)))))
    (lambda (lambda-exp)
      (apply append (collector lambda-exp)))))

(define (make-cond-clause-collector dont-want-it want-it)
  (lambda (lambda-exp)
    (list-transform-negative (map (lambda (cond-clause)
                                    (if (dont-want-it (clause/predicate cond-clause))
                                        '()
                                        (want-it      cond-clause)))
                                  (cond/clauses (lambda/body lambda-exp)))
      null?)))
```

Using these collectors, we can define the action. It begins by let binding a set of viewcode expressions that will be useful later on. It then calls replace, to replace dispatch-expression's lambda expression with our hash-based template. Notice that we bind exp-type-symbol to the expression that computes the type symbol (from type?'s implementation). By binding this first, we avoid inadvertently capturing any free variables the expression may contain.

```
(define action/dispatch-expression
  (make-explained-action
    (lambda (vform context vc-exps)
     (let* ((dispatch-exp-value     (vc-exps/dispatch-val vc-exps))
            (dispatch-formal        (first (lambda/formals dispatch-exp-value)))
            (type?-lambda           (vc-exps/type?-lambda vc-exps))
            (type?-formals          (lambda/formals type?-lambda))
            (type?-lambda-comb-args (combination/args (lambda/body type?-lambda))))
       (replace dispatch-exp-value
                `(let ((exp-type-symbol (lambda (,(second type?-formals))
                                          ,(if (var-bound-by? (first type?-lambda-comb-args)
                                                              (first type?-formals) context)
                                               (second type?-lambda-comb-args)
                                               (first  type?-lambda-comb-args))))
                       (dispatch-table  (make-symbol-hash-table)))
                   (for-each (lambda (symbol process-exp)
                               (hash-table/put! dispatch-table symbol process-exp))
                             (list ,@(collect-type-symbols  dispatch-exp-value))
                             (list ,@(collect-process-procs dispatch-exp-value)))
                   (lambda (,dispatch-formal)
                     (let ((process-exp (hash-table/get dispatch-table
                                                        (exp-type-symbol ,dispatch-formal)
                                                        #f)))
                       (if process-exp
                           (process-exp ,dispatch-formal)
                           (begin ,@(collect-else-consequents dispatch-exp-value)))))))))
   "converting dispatch-expression into a O(1) time dispatch"))
```

4.9.5 Coupling/Coupler Production

The goal in this step is to produce a coupler. We begin by producing a dispatcher. Given our predicates and action, we combine them into a dispatcher. Within this combination, we create a compound predicate using the ViewForm combiner `combine-vforms-conjunctive`. This function simply combines vforms using the previously defined control structure `*conjunctive-control-structure*`. The code that implements this dispatcher is given below.

```
(define dispatcher/dispatch-expression
  (make-simple-dispatcher (combine-vforms-conjunctive predicate/usual-integrations
                                                      predicate/no-mutations
                                                      predicate/type?
                                                      predicate/dispatch-expression)
                          action/dispatch-expression))
```

Before writing our coupler rule, we implement its control structure. This control structure follows the same pattern as the one given earlier in Section 4.4.5. It first creates a context from a set of given files (since we will be invoking views over the combined expressions). It then ensures that the alpha and liar views are computed. The vforms are then invoked on a two-element viewcode list consisting of the viewcode expressions for `dispatch-expression`'s value (e.g., `(lambda (exp) (cond …))`) and for `type?`'s value (e.g., `(lambda (type-keyword? exp) …)`. If the coupling was successful, it is registered.

```
(define vc-exps/dispatch-val car)
(define vc-exps/type?-lambda cadr)

(define (make-dispatch-exp-control-structure file1 . rest-files)
  (lambda (vforms)
    (lambda (vform context vc-exps)
      (let ((context (or context (apply make-context-from-files (cons file1 rest-files)))))
        (ensure-liar-view-computed!  context)
        (ensure-alpha-view-computed! context)
        (let ((dispatch-expression-def (top-lvl-name->def 'dispatch-expression context))
              (type?-def               (top-lvl-name->def 'type?              context)))
          (if (and dispatch-expression-def type?-def)
              (if (for-all? vforms
                    (lambda (vform)
                      (do-vform vform context (list (define-var/value dispatch-expression-def)
                                                    (define-var/value type?-def)))))
                  (register-coupling! context vform)))
          context)))))
```

We can now write the coupler rule. It combines the control structure, the dispatcher, and an explanation combiner.

```
(define dispatch-exp-rule
  (combine-vforms (make-dispatch-exp-control-structure
                    (fully-qualify *dispatch-exp-dir* "linear-cond")
                    (fully-qualify *dispatch-exp-dir* "types"))
                  (verbose-expl-combiner "dispatch-expression example rule")
                  (list dispatcher/dispatch-expression)))
```

The rule can be invoked by evaluating the expression `(do-vform dispatch-exp-rule #f '())`. When this was done, ViewForm generated the code in Figure 1-8 for `dispatch-expression`.

4.9.6 Invalidation/Recoupling

Suppose that, after ViewForm generates the coupled code in Figure 1-8, we were to modify the implementation of `type?` to the faster version in Figure 1-5. At this point, our coupled code is now invalid. To perform the recoupling step, we simply invoke the view invalidation/recoupler: `(vir!)`. This causes ViewForm to produce the following code for `dispatch-expression`:

```
(define dispatch-expression
  (let ((exp-type-symbol (lambda (expression) (fast-exp-type expression)))
        (dispatch-table (make-symbol-hash-table)))
    (for-each (lambda (symbol process-exp)
                (hash-table/put! dispatch-table symbol process-exp))
              (list (quote literal)    (quote java-name)     (quote new)
                    (quote dot)        (quote call)          (quote cast)
                    (quote instanceof) (quote built-in-expr) (quote assignment))
              (list process-literal    process-java-name     process-new
                    process-dot        process-call          process-cast
                    process-instanceof process-built-in-expr process-assignment))
    (lambda (exp)
       (let ((process-exp (hash-table/get dispatch-table (exp-type-symbol exp) ())))
        (if process-exp (process-exp exp) (begin (error "Unknown Type: " exp)))))))
```

4.10 User Interaction

Vforms can interact with external entities such as the user. In its simplest form, user interaction can be implemented using Scheme's native I/O facilities. While this implies that any vform can interact with the user, predicates are the vform most likely to benefit from such interaction. Predicates in dispatchers, for example, could ultimately ask the user to approve or disapprove a precondition if the available views are not precise or fast enough to make the determination.

Predicates that compute views can also ask the user for assistance. For example, a program termination view, though generally uncomputable, could ask questions such as, "does this loop terminate?" This kind of assistance is necessary for complete backwards compatibility, to allow users to "vouch" for program properties that are part of a coupling's preconditions, but are not generally computable. I do not explore this kind of user interaction in this dissertation, but find it necessary to support and to mention it.

In order to better automate the view invalidation/recoupler, user interaction that is not necessary should be kept to a minimum. One way to do this is to cache or store a user's responses in a view. A user can then modify that view as necessary, and the view can ask the user for a response if no previous response exists or if the previous response in invalid. Ideally, the user's responses can be checked by the view, although this will not always be possible.

4.11 Complexity Layering

Implementing an abstraction model that addresses the implementation coupling problem is not a simple task, especially when simplicity in the model pushes complexity into the implementation. My approach to this issue is based on a concept familiarized in [55] and later in Intrigue[59]. Intrigue was designed so that the less a programmer wanted to change the Intrigue implementation, the less complex the code that had to be written. I refer to this as *complexity layering*. That is, the least complex interfaces are available at the highest abstraction layer, whereas the most complex interfaces are layered more deeply inside. The least complex ViewForm interfaces offer high level, commonly used, but somewhat limited functionality (i.e., `make-predicate/find-declaration`). The deeper, more complex, interfaces offer low level, infrequently used, but more powerful functionality (i.e., `make-predicate`, `make-view`).

ViewForm was designed as a complexity-layering implementation. In part, complexity layering is provided by a coupler and view library. The goal is to make this library of high-level constructs ample enough to suffice in most situations. Complexity layering is also provided by the ability to decompose and reuse existing vforms. Another supporting design aspect is the ability to work at varying layers of abstraction (couplers, rules, dispatchers, predicates, actions). In addi-

tion, by allowing vform composition and abstraction, language experts can build low-level constructs that can be combined by other users with less expertise into higher-level, more abstract constructs. These higher-level constructs can then be used by the end users, insulating them from the knowledge wielded by the language experts. This "build once, use many times" approach together with the "make it easier to use, even if harder to implement" approach means that a lower-level vform's complexity and development cost can be amortized over many uses. Nevertheless, complexity in ViewForm can be categorized into four layers:

a) Writing views
b) Writing predicates/actions (using views)
c) Writing rules/dispatchers (using predicates and actions)
d) Writing couplers (using rules and dispatchers)

Layer a is the most complex[***], whereas layer d is the simplest. Since views are the most complex to design and implement, a view should be designed to be useful in many circumstances. Couplers, on the other hand, are more likely to be specific to a given coupling. Since couplers can be composed from existing rules, programmers can reason about their construction at a higher level of abstraction. The benefits of this complexity layering structure have already helped in the development of the simple examples presented thus far. The benefits will also help in the development of the more complex examples that follow in Chapter 5.

[***] As more research is devoted to program analysis algorithms, it will become more likely that a view can be implemented directly from a published paper. In such a case, writing views will be much less complex than in the general case.

Chapter 5

Examples and Analysis

Having discussed the view-based abstraction model, methodology, and an implementation, the next step is to investigate the degree to which they solve the implementation-coupling problem and achieve the desiderata from Chapter 2. With this goal in mind, this dissertation examines three examples that benefit from implementation coupling. The first example is code that simulates an amorphous computing substrate[7][†††], the second is code for computing a conditional probability for a particular pedigree problem [83], and the third is the ViewForm implementation itself. For each example, the desired couplings are presented and then expressed using ViewForm, the view-based abstraction model, and the view-based abstraction methodology. Afterwards, the ViewForm code is quantified, analyzed and evaluated. This chapter concludes with a discussion of further issues, ideas, motivations, and lessons learned.

5.1 Testing Methodology

The goal of the testing methodology is to provide a consistent approach for evaluating whether view-based abstraction successfully preserves modularity in the presence of implementation couplings. This approach consists of two aspects: a testing process and evaluation metrics. The testing process is a set of steps applied to each example. The evaluation metrics are used to measure success. There are two such metrics. The first is whether ViewForm can successfully generate coupled code (or its equivalent) that satisfies the programmer's criteria.[‡‡‡] The second is whether the desiderata from Chapter 2 are met. The former is a mostly objective question: either ViewForm generates the desired coupling or it does not. Aspects of the latter metric, however, can be somewhat more subjective. The process and the metrics are discussed below.

[†††] This substrate is affectionately known as "gunk".

[‡‡‡] An equivalent piece of code is consistent with the reasons the programmer wanted the coupling in the first place.

5.1.1 Testing Process and Evaluation Metrics

The testing process consists of a series of steps applied to each of the three examples. These steps are:

- Present the uncoupled code, the coupling code, and the desired coupled code
- Implement the coupling using the view-based abstraction methodology (i.e., the six steps)
- Run the resulting coupler and present the generated coupled code
- Modify either the uncoupled code or the coupling code
- Invoke the view invalidation/recoupler and present the regenerated, coupled code
- Count the number of predicates, actions, dispatchers, rules, couplers, and views needed to implement the coupling
- Count the lines of uncoupled code modified by the actions
- Count the lines of uncoupled code that actions depended upon but did not modify
- Measure the amount of time needed to couple the code
- Measure the amount of time needed to recouple the code
- Measure the amount of time needed to compile the code under MIT Scheme

The results of this process are used to assess the two evaluation metrics. The first metric is whether ViewForm generated the desired code. For this metric, the coupled code is examined with respect to the criteria for wanting the coupling to begin with. For instance, in the dispatch example, the criterion was performance. Specifically, a linear-time process was to be converted to an expected constant-time process by using hash tables. Given this criterion, the code in Figure 1-4 is equivalent to the code in Figure 1-8, even though they are not syntactically the same.

The second evaluation metric is whether the example demonstrated the properties of backwards compatibility, incrementality, language independence, ease of understanding and usability, and amortizable time savings. Let us examine each of these properties individually.

The evaluation of backwards compatibility will be assessed using the list from Section 2.3.1. If the coupling is consistent with all five prohibitions, then backwards compatibility will be deemed successful.

For this dissertation, incrementality will be assessed by comparing the scope of the desired coupling to the scope of the actual coupling. For example, in the dispatch example, the predicates depended on the same code as the preconditions, and the actions on the same code as the desired modifications. To demonstrate incrementality, a coupling must not require the application of view-based abstraction to non-related parts of the program or to other existing implementation couplings.

Language independence in view-based abstraction cannot be easily tested via the examples because they are all written in MIT Scheme. Instead, this property is gleaned by virtue of the view-based abstraction model and methodology in Chapter 3 not depending on any Scheme language property or feature. While ViewForm itself depends on Scheme, view-based abstraction does not preclude implementations of ViewForm in other languages for other languages. Such implementations are a topic for future work.

Ease of understanding and usability are subjective properties. To better evaluate these properties with respect to an example, some aspects of each implementation coupling are quantified. One such aspect is the number of lines of ViewForm code per coupling construct. This quantity corresponds to the amount of work needed to write the construct. It can also be used as a first-order measure of the ease of understanding ViewForm; the assumption being that the number of lines of code is an upper bound on the number of calls to ViewForm functions needed to perform the coupling. A coupling that needs to call few ViewForm functions is more likely to require less overall understanding of ViewForm than a coupling that needs to make many calls to ViewForm. Of course, the complexity level of each call must also be taken into account. Another useful measure is to compare the number of lines of code needed to implement a coupling to the number of lines of code actually modified or depended upon by a coupling. This ratio helps to normalize otherwise absolute figures. This ratio is useful for comparing the amount of work needed to implements different couplings. While it is clear that people will disagree on what these numbers really mean, these numbers do provide useful information for discussing, comparing, and evaluating couplings.

To estimate the amount of time saved by using view-based abstraction to perform an implementation coupling, the total amount of work done to implement, carry out, and regenerate the coupling must be determined. As discussed above, the number of lines of code needed to write a coupling reflects the amount of work applied to implement it. This measure is a more objective quantity than timing how long a programmer would need to implement the ViewForm code. For example, measuring the raw amount of time I personally put into implementing the coupling is not meaningful since I am quite familiar with ViewForm.

One time measurement that is useful is the amount of time needed for ViewForm to perform a coupling and regenerate the coupling. This can be used to evaluate how practical ViewForm is by comparing it to how long it takes to compile the uncoupled code. The timing numbers in the sections below were taken on a Pentium 133 (no MMX), with a 512K pipelined-burst secondary cache, and 64 megabytes of main memory (i.e., no observed swapping). The Scheme used was MIT Scheme version 7.4.2, compiled using Microsoft Visual C++ 4.2.

```
(declare (usual-integrations)
         (integrate-external "gunk"))

(define (gunk:make-particle int:UID x y)
  (vector                   int:UID x y
                            gunk:initial-value
                            gunk:null-neighbors ; Initially, a list; later vect
                            gunk:non-boundary))
;; Initials
(define-integrable gunk:initial-value     0.0 )
(define-integrable gunk:initial-value/top 1.0 )
(define-integrable gunk:null-neighbors    '() ) ; Initially a list; later vect
(define-integrable gunk:boundary          true)
(define-integrable gunk:non-boundary     false)

;; Access
(define-integrable (gunk:particle-UID         particle) (vector-ref particle 0))
(define-integrable (gunk:particle-x           particle) (vector-ref particle 1))
(define-integrable (gunk:particle-y           particle) (vector-ref particle 2))
(define-integrable (gunk:particle-value       particle) (vector-ref particle 3))
(define-integrable (gunk:particle-neighbors   particle) (vector-ref particle 4))
(define-integrable (gunk:particle-special-tag particle) (vector-ref particle 5))

;; Mutation
(define-integrable (gunk:set-particle-value!          particle   new-value)
  (                                vector-set!    particle 3 new-value))

(define-integrable (gunk:add-new-particle-neighbor!   particle   new-neighbor)
  (                                vector-set!    particle 4
                                                  (gunk:add-new-neighbor
                                                            new-neighbor
                                                            (gunk:particle-neighbors
                                                             particle))))
(define-integrable (gunk:finalize-particle-neighbors! particle)
  (                                vector-set!    particle 4
                                                  (gunk:finalize-neighbors
                                                   (gunk:particle-neighbors particle))))

(define-integrable (gunk:set-particle-special-tag! particle   new-value)
  (                                vector-set! particle 5 new-value))

(define-integrable (gunk:add-new-neighbor new-neighbor neighbors  ) ; Initial
  (cons                             new-neighbor neighbors  ))

(define-integrable (gunk:finalize-neighbors           neighbors  ) ; Finalize
  (list->vector                                       neighbors  ))

(define-integrable (gunk:neighbors-count              neighbors  ) ; Finalized
  (      vector-length                                neighbors  ))

(define-integrable (gunk:get-neighbor                 neighbors N) ; Finalized
  (      vector-ref                                   neighbors N))

(define (gunk:particle-neighbors-UIDs                             particle)
  (vector-map->vector gunk:particle-UID (gunk:particle-neighbors particle)))

(define (gunk:particle-neighbors-count                            particle)
  (gunk:neighbors-count               (gunk:particle-neighbors particle)))

(define (gunk:particles-neighbors-counts                          particles)
  (       map         gunk:particle-neighbors-count               particles))
```

Figure 5-1 - Particle Data Abstraction Implementation, adapted by Blair from code by Abelson

5.2 Amorphous Computing Simulation

The amorphous computing simulation example comes directly from Blair's work on Descartes.[7][§§§] This example simulates a large set of proximate and spatially constrained but otherwise independent, SPMD computational entities. The simulation computes a diffusion gradient over all the entities based on some initial boundary conditions.

Blair's objective was to dynamically profile and specialize the amorphous computing simulator in order to increase performance. Through profiling, Blair found that he could produce an 8x speedup by replacing functions in the data structure shown in Figure 5-1 with the specialized versions shown in Figure 5-2. The speedup results from the use of floating-point vectors (flovecs), which require less effective floating-point number boxing and unboxing and less garbage production. This modification is thus a data representation shift.

```
(define-integrable gunk:initial-value
  (let ((flovec (flo:vector-cons 1)))
    (flo:vector-set! flovec 0 0.0)
    flovec))

(define-integrable (gunk:set-particle-value! particle new-value)
  (flo:vector-set! (vector-ref particle 3) 0 new-value))

(define-integrable (gunk:particle-value particle)
  (flo:vector-ref  (vector-ref particle 3) 0))
```

Figure 5-2 - Optimized Particle Value Implementation

The original particle value interface allows a particle's value to be represented by an arbitrary data structure. This specification is honored by the uncoupled code in Figure 5-1. In the Figure 5-2 specialized case, however, a particle's value can be represented only by a floating point number (float). This representation shift therefore violates the original interface since the shift is highly conditionalized on the simulator's implementation using floating point numbers to represent particle values. An update to the simulator that used a different representation would result in an invalidating implementation with respect to this implementation coupling. Blair's system cannot produce this kind of specialization in a semantics-preserving way specifically because the specialization results in an implementation coupling.

5.2.1 Amorphous Computing Simulator Background

Blair took the original simulator code given to him and modified it for personal aesthetic and comprehensibility reasons. The resulting simulator consists of 1,280 lines of code (~1,100 non-comment lines) in two files that call selected functions in 6,000 total lines of code spread

[§§§] The simulator code was written by Hal Abelson, and was subsequently modified by Blair.

```
(define-integrable gunk:initial-value
  ((lambda (initial-float flov)
     (flo:vector-set! flov 0 initial-float)
     flov)
   0. (flo:vector-cons 1)))

(define-integrable gunk:set-particle-value!
  (lambda (particle new-value)
    (flo:vector-set! ((lambda (particle)
                        (vector-ref particle 3))
                      particle)
     0 new-value)))

(define-integrable gunk:particle-value
  (lambda (particle)
    (flo:vector-ref ((lambda (particle)
                       (vector-ref particle 3))
                     particle)
     0)))
```

Figure 5-3 - ViewForm Output

over 14 library files. These 14 files reference (but for the simulator, do not call) functions in over 20 other files containing over 10,000 lines of code. As demonstrated below, however, the core simulator code ends up being the only code that must be contextualized to produce the specialization coupling. This finding is consistent with my experience that implementation coupling is usually needed in proportionately few and relatively specific parts of an application.

I neither wrote nor modified the simulator implementation, and I did not influence the choice of specializing optimizations discovered by Blair. The amorphous computing simulator example can therefore be used to explore view-based abstraction and ViewForm on code and on an implementation coupling that had no influence on view-based abstraction's design or implementation (and vice versa). This makes the simulator example a completely independent example.

5.2.2 Building the Coupling

Blair considered the aforementioned 8x speedup so great that it outweighed the subsequent loss of modularity caused by the implementation coupling. Under view-based abstraction, however, this specialization can work without precluding modularity. Without the specialization, there is an interface coupling from the simulator module to the particle data abstraction module. Since the simulator is layered over the particle data abstraction, this corresponds to Figure 1-1 (the simulator module is M_1 and the particle data abstraction module is M_2). The Figure 5-2 specialization, however, induces an implementation coupling from M_2 to M_1. While this is the reverse direction of what is illustrated in Figure 1-7, it does not pose any problem since the view-based abstraction model, methodology, and implementation do not depend on any coupling directionality. To perform this coupling, let us use the implementation coupling steps (i.e., I-VI) and the view-based abstraction methodology steps (i.e., i-vi) from Chapter 3.

Our goal in the sections that follow is to build a coupler that performs the desired specialization modification, after validating the modification's preconditions. The approach will be to build bottom up, combining the vforms along the way. We will build predicates, actions, and explanations, then dispatchers, rules, and a coupler.

5.2.3 Preconditions and Predicates

Before building a set of predicates, we must determine the preconditions under which the particle value representation can be shifted to use flovecs. These preconditions, discussed with Blair, are given below:

A. The particle value data abstraction interface is fixed
B. The particle value data abstraction is not violated
C. Floats do not have identity with respect to `eq?`
D. Predefined Scheme procedures are not redefined
E. The modules to which the particle data abstraction is available are known
F. A particle's initial particle value (i.e., `gunk:initial-value`) is a float
G. New particle values passed to `gunk:set-particle-value!` are all floats

A programmer is responsible for developing preconditions like the ones above. Usually, a programmer has no need for making them weaker or less fragile. While making preconditions less fragile is not absolutely necessary, it does make the resulting coupler more useful and flexible. The degree of fragility can vary according to the specifics of the situation and the pragmatics of computing the views needed by the predicates that will implement the preconditions.

Preconditions A-G are discussed in detail below, for the purpose of relating them to view-based abstraction and ViewForm. The predicates implementing these preconditions are also presented and discussed.

```
(define predicate/usual-integrations
  (make-predicate/find-declaration 'usual-integrations))

(define predicate/returns-inexact
  (make-predicate/check-ret-type *inexact-type* "expression returning a float"))

(define predicate/flov-selector
  (make-predicate/exp-type type/define-integrable? "define-integrable expressions"))

(define predicate/flov-mutator
  (make-predicate/check-proc-param-types (list (cons second predicate/returns-inexact))))
```

Figure 5-4 - Simulator Example Predicates

Conditions A and B are typically assumed to be true, as they are for the simulator example. They are presented for this example for completeness. Even if these preconditions were not assumed true, condition A can be checked to some extent: the parameter list itself, return value types, and argument types can be checked statically via data-flow analysis. If necessary, these (necessarily conservative) static checks could be supplemented by statically inserted "guard" conditional expressions that dynamically confirm object types. While expensive, these dynamic

checks would provide a higher, if not complete, level of safety. As is usually the case, it is up to the programmer, based on the specifics of the situation, to make the appropriate tradeoffs between performance and varying degrees of safety. Other ways of checking Condition A under view-based abstraction are discussed in Section 6.6.2.

In Scheme, precondition B is difficult to automatically validate in general. Precondition B is easier to validate in languages that provide statically-checkable interface semantics such as CLU[61]. While a violation of precondition B could be harmful with or without view-based abstraction, a programmer may not necessarily need B to be true. This may be the case when using view-based abstraction on code that contains previously existing abstraction violations of some sort. Legacy code can suffer from this problem, for example. In these cases, the preconditions may need to become more complex, constraints on future modifications may need to be enforced, or safety may be put at risk.

For instance, in the simulator example, if precondition B was not assumed true, every abstraction violation to the particle value field of a particle data structure would need to be found, to determine whether it uses floats. In addition, all data structures passed to `gunk:particle-value` and `gunk:set-particle-value!` would also need to be found and checked to ensure that the particle value field was provably a float. Supplemental dynamic checks (and possibly type coercions) might also be needed, if a static analysis view was not precise enough for a predicate to prove that all abstraction violations of the particle data structure used floats to represent particle values.

For the simulator example, however, while violating precondition B is a semantic problem and possibly a performance problem, it is not actually a safety problem. This is because in MIT Scheme, a 1-element flovec is operationally (but not semantically) equivalent to a float.

Precondition C is true under the IEEE Scheme standard[49], meaning that a program using `eq?` on floats is not inherently portable. Precondition C, however, is not necessarily true in MIT Scheme. As mentioned above, floats and 1-element flovecs are operationally equivalent. Semantically, however, they differ with respect to mutability. A flovec is mutable, whereas a float is considered atomic data and thus is not. This means that flovecs must have identity, and hence, certain shared floats also have identity and `eq?` equality under MIT Scheme. For the simulator example, however, we choose to follow the standard and assume that `eq?` on floats is an implementation-dependent operation whose result is unspecified.

Under other circumstances, the identity issue would need to be addressed. For instance, we might want to shift representations from one data structure to another with differing, but specified, identity characteristics. In these cases, an alias analysis view may be needed by the predicate validating the analog of precondition C to ensure the value is not being tested for `eq?`-like

equality. In the worst case, however, the alias analysis will not be precise enough to validate an otherwise valid precondition. For these cases, user interaction would likely be required.

Precondition D is necessary to ensure that the flovec primitives we insert will be referencing the correct primitive procedures (i.e., `flo:vector-ref`, `flo:vector-cons`, etc.). If this precondition were not true, the desired primitive procedures would have to be accessed some other way, such as using the MIT Scheme special form `access`. In MIT Scheme, precondition D can be generally checked by looking for the `usual-integrations` declaration. This user-specified declaration allows the compiler to open code calls to primitives. In effect, this nullifies any primitive name redefinitions. A predicate that expresses this precondition, `predicate/usual-integrations`, is given in Figure 5-4. We write this predicate using the ViewForm library function `make-predicate/find-declaration`[****]. This function was previously used in the dispatch example. The function takes a declaration symbol and returns a predicate that looks for such a declaration in a context.

Precondition E determines which files must be passed to ViewForm for analysis. The filenames will be passed to our coupler, which will then create the appropriate contexts to be analyzed. In our example, it turns out that the simulator implementation's file is the only one layered over the particle data abstraction. If other files were to begin using the particle data abstraction, they would also have to be passed to the coupler we will write. A programmer using black-box abstraction will likely thoroughly and manually check precondition E, since the core motivation for the specializing modification is the observation that *every* use of `gunk:set-particle-value!` uses a float to represent a particle's new value.[††††]

If the programmer is not able to satisfy precondition E, there are several alternatives. The simplest, but potentially the costliest, is to pass all available expressions in the application to the coupler (as a context(s)). A data-flow analysis can then be used to conservatively estimate precondition E, and alert the programmer to any problems. A less costly, but weaker test is to check all available expressions for references to the particle value interface functions. An escape analysis view can also help the programmer make the determination, even if view-based abstraction were not being applied.

Precondition F can be determined via a data-flow view on `gunk:initial-value`. This assumes that `gunk:make-particle` will use `gunk:initial-value` to reference the initial particle value. Let us assume this is the programmer's intent, since there is little benefit to otherwise naming the initial value. This assumption does, however, constrain the way future changes to the initial value can be made. In particular, this assumption requires that changes to the initial value to be made to

---

[****] The code for `make-predicate/find-declaration` is given in Appendix B.

[††††] This was the key insight that Blair's profiler revealed.

`gunk:initial-value`'s value and not to `gunk:make-particle`. While it can be argued that this assumption adheres to the programmer's intent and style, several alternatives are discussed below, in case this assumption was determined not to be the case.

Precondition F's predicate must ensure that `gunk:initial-value` can name only objects that are floats. We implement this using the predicate `predicate/returns-inexact`, given in Figure 5-4. We will later ensure that this predicate receives the viewcode expression representing `gunk:initial-value`'s value. We implemented `predicate/returns-inexact` using ViewForm's higher-level predicate constructor `make-predicate/check-ret-type`. This predicate constructor is parameterized on a type and an explanation string. The returned predicate checks every viewcode expression it is given to ensure that all values returned by each expression are consistent[‡‡‡‡] with the given type. For the interested reader, the code for `make-predicate/check-ret-type` is given in Appendix B.

One alternative to our choice for expressing precondition F is to check the actual place in the `gunk:make-particle` definition where the initial particle value is computed. In fact, we could use the same predicate defined above to do this check, by instead passing it the appropriate expression in `gunk:make-particle`. Unfortunately, this alternative constrains the particle value to be the fourth argument of a `vector` expression. Thus, this alternative does not result in a more robust predicate.

Another alternative, the safe and more flexible but more costly alternative, is to ignore `gunk:initial-value` and instead insert a single run-time check that examines the result of each call to `gunk:make-particle`. This dynamic type check simply ensures that a particle's initial value is always a float. This alternative also necessitates writing an action that inserts run-time code (into `gunk:make-particle`) that replaces the initial float with a flovec. While this is straightforward to do, this alternative was not chosen for two reasons. The first is the high overhead in the dynamic test and float replacement. The second is that the specialization modification's purpose is to reduce the amount of boxing and the number of heap-allocated floats that become garbage. While completely safe, this alternative does little to reduce the amount of boxing and garbage being created.

A fourth simple, but semantically inelegant alternative is to assume the functional equivalence between floating point numbers and 1-element flovecs. Under this alternative, no change to the initial value is necessary, so long as it is always an integrated float or a 1-element flovec. A predicate can ensure that it is always integrated and a dynamic type-check can ensure the value

[‡‡‡‡] Consistent means that the return value is of the given type or of a subtype of the given type.

is always a float or 1-element flovec. As these alternatives illustrate, there is no general "right" way to implement preconditions as predicates.

Condition G can be checked by examining `gunk:set-particle-value!`'s parameter list. `gunk:set-particle-value!` must always be passed a float as the second argument. This precondition is the fundamental basis of the motivation to perform the specialization modification. We validate this precondition using the predicate `predicate/flov-mutator`, given in Figure 5-4. This predicate is produced from the general ViewForm predicate constructor `make-predicate/check-proc-param-types`. This predicate constructor takes a type specification list, which contains pairs of parameter list selectors and vforms. The type specification list used by `predicate/flov-mutator` requires that the `second` parameter to `gunk:set-particle-value!` must satisfy the predicate `predicate/returns-inexact` (also shown in Figure 5-4).

We need one last predicate to validate the actual modification. To increase our predicate's robustness, our code modification should not depend on the original selector's or mutator's implementation. As shown in Figure 5-2 and Figure 5-3, we do this by copying the original selector's code into the new mutator. Before copying the original selector's code, however, we need to ensure that copying it is safe. This is what the final precondition is for. If the code contains side effects, for example, copying it might not be safe. Instead of performing a side-effect analysis, however, we can leverage on the fact that the original definition of `gunk:particle-value` is defined using `define-integrable`. This MIT Scheme special form instructs the compiler to substitute the body of the definition anywhere the name appears, which is exactly the property we desire. The predicate `predicate/flov-selector` in Figure 5-4 checks for this condition. It expects to be passed `gunk:particle-value`'s defining expression, which we will arrange for later. This predicate calls the ViewForm predicate constructor `make-predicate/exp-type`, which simply checks the expression type of a given viewcode expression. This predicate constructor is widely used in view dispatchers, where the appropriate action on an expression is determined by the expression's type.

These four predicates are the only ones we need. They must all be true before any program modifications are performed.

5.2.4 Actions

We turn next to writing actions that express the modifications necessary to shift the particle value representation to flovecs. One goal is to reduce the actions' scope and another is to increase their robustness by making them depend as little as possible on the original selector, mutator, and initial value implementations. This will allow a maintenance programmer to modify

the original mutator's and selector's implementation without invalidating the action's modifications.

Before writing the action, we must decide upon a set of program modifications that will transform the selected functions in Figure 5-1 to those in Figure 5-2. These modifications desired by Blair can be summarized as follows:

a) Change the particle's initial value to a flovec
b) Change `gunk:set-particle-value!` to modify the flovec
c) Change `gunk:particle-value` to return a float from the flovec

Our strategy is to express three actions corresponding to Changes a-c. To limit the scope in Change b, we will require access to the original selector's code. For this reason and for symmetry, we will write our actions to assume they will each be passed all three of the viewcode expressions respectively passed to our predicates. The actions can then select which expressions to replace, and can build the replacements based on any of the three original expressions. Due to this special implementation decision, we use `make-explained-action` to construct our actions. `make-explained-action` is a lower-level action constructor, and does not provide as much functionality as the predicate constructors we used earlier. `make-explained-action` is like `make-action` (section 4.4.3) except that it takes an explanation string that will be prepended with the string "`action for `" and then printed to the standard output when the action is explained.

```
(define action/flov-init-val
  (make-explained-action
   (lambda (vform context vc-exps)
     (replace (vc-exps/init-val vc-exps)
              `((lambda (initial-float flov)
                  (flo:vector-set! flov 0 initial-float)
                  flov)
                ,(vc-exps/init-val vc-exps) (flo:vector-cons 1))))
   "gunk:initial-value's flonum expression with floating-point vector"))

(define action/flov-mutator
  (make-explained-action
   (lambda (vform context vc-exps)
     (let ((selector-val (full-copy (define-var/value (vc-exps/selector-def vc-exps)))))
       (replace (vc-exps/mutator-val vc-exps)
                `(lambda (particle new-value)
                   (flo:vector-set! (,selector-val particle) 0 new-value)))))
   "flovec'ing gunk:set-particle-value!"))

(define action/flov-selector
  (make-explained-action
   (lambda (vform context vc-exps)
     (let ((selector-val (define-var/value (vc-exps/selector-def vc-exps))))
       (replace selector-val `(lambda (particle)
                                (flo:vector-ref (,selector-val particle) 0)))))
   "flovec'ing gunk:particle-value"))
```

Figure 5-5 - Simulator Example Actions

We begin with an action for performing Change a. We base the action on a code template that creates a 1-element flovec, sets the flovec's value to `gunk:initial-value`'s original value, then returns the flovec. This template is implemented in the action `action/flo-init-val`, given in Figure 5-5. The selector `vc-exps/init-val` returns the appropriate viewcode expression from the passed in viewcode expression list, `vc-exps`. We define this selector later.

Changes b and c are performed by `action/flov-mutator` and `action/flov-selector`, respectively. These actions are given in Figure 5-5. The strategy for the particle value mutator is to substitute a code template that uses the original particle value selector code to pull the flovec out of the particle object, then use `flo:vector-set!` to mutate that flovec. For the particle value selector, we use a code template that uses the original particle value selector code to pull the flovec out of the particle object, then uses `flo:vector-ref` on the flovec. Note that in either case, the action does not depend on the current particle data structure implementation. If a maintenance programmer later modifies the selector or mutator, the actions will remain valid modifications. As discussed earlier, these actions do depend on the original selector code being copyable. If this were not the case (i.e., `gunk:particle-value` was not `define-integrable`), a different action for the mutator would be necessary. One straightforward and safe approach in this case would be to rename the original selector and have a new mutator and new selector call the original code to pull out the flovec. This, however, would add an extra, full procedure call to each selector and mutator call. Another approach is to lexically capture a single definition of the original `gunk:particle-value` within a closure, then use mutation to appropriately set the top-level mutator and selector names to new procedures that use the original selector code to pull the flovec out of the particle object.

5.2.5 Dispatcher and Coupler

We can now combine our predicates and actions into a dispatcher. The dispatcher will espouse the relationship between our predicates and our actions: if the predicates are all true, then the actions are all invoked. This is accomplished by `dispatcher/gunk-flov`, given in Figure 5-6. We use two higher-level vform combinators: `combine-vforms-conjunctive` and `combine-vforms-conjunctive-1-to-1`. Each takes any number of vforms and returns a single vform with a specific control structure. `combine-vforms-conjunctive`'s control structure runs each vform sequentially from left to right on every viewcode expression, but stops and immediately returns `#f` if any vform returns `#f`. `combine-vforms-conjunctive-1-to-1` is similar, but runs the n^{th} vform on the n^{th} viewcode expression. These combiners and the resulting dispatcher cleanly describe what the dispatcher does. The dispatcher first looks for a `usual-integrations` declaration. If found, it sequentially checks the initial value, the mutator, then the selector preconditions. When the predicates validate the preconditions, the actions are invoked to modify the program.

Before combining the dispatcher into a coupler, we still have a few things left to do. These items will be handled by the coupler's control structure:

d) Create the proper contexts
e) Ensure the proper views are computed
f) Gather the viewcode expressions to be passed to the dispatcher
g) Register the resulting coupling

Steps d-g are generally performed by a coupler's control structure, as in the dispatch example. For the simulator example, we implement these steps in the control structure constructor given in Figure 5-6. This constructor takes a set of filenames and returns a control structure. Step d is performed in this control structure using `make-context-from-files`, which takes a series of files as arguments and returns a context containing all of the file expressions. The implementation also

```
(define dispatcher/gunk-flov
  (make-simple-dispatcher
   (combine-vforms-conjunctive predicate/usual-integrations
                               (combine-vforms-conjunctive-1-to-1 predicate/returns-inexact
                                                                  predicate/flov-mutator
                                                                  predicate/flov-selector))
   (combine-vforms-conjunctive action/flov-init-val
                               action/flov-mutator
                               action/flov-selector)))

(define-integrable vc-exps/init-val     car)
(define-integrable vc-exps/mutator-val  cadr)
(define-integrable vc-exps/selector-def caddr)

(define (make-gunk-vform-control-structure acs-file . acs-files)
  (lambda (vforms)
    (lambda (vform context vc-exps)
      (let ((context (or context (apply make-context-from-files (cons acs-file acs-files)))))
        (ensure-liar-view-computed!  context)
        (ensure-alpha-view-computed! context)
        (let ((vc-exps
               (list (define-var/value (top-lvl-name->def 'gunk:initial-value       context))
                     (define-var/value (top-lvl-name->def 'gunk:set-particle-value! context))
                                       (top-lvl-name->def 'gunk:particle-value      context))))
          (if (for-all? vforms
                (lambda (vform) (do-vform vform context vc-exps)))
              (register-coupling! context vform)
              (format #t "~%Flovectorization failed!"))
          context)))))

(define acs-flov-rule
  (combine-vforms (make-gunk-vform-control-structure
                   (fully-qualify *gunk-dir* "gunk")
                   (fully-qualify *gunk-dir* "gunk-particles"))
                  (verbose-expl-combiner "flo-vectorizing gunk example")
                  (list dispatcher/gunk-flov)))
```

Figure 5-6 - Simulator Example Dispatchers and Other Rules

leaves open the possibility that a previously constructed context will be passed in instead.

`ensure-liar-view-computed!` and `ensure-alpha-view-computed!` ensure that the appropriate views are available. We use the ViewForm liar view and alpha view, described in Section 4.6. Since our actions do not require updated liar or alpha information, there is no need to recompute any views during the code modifications.

The viewcode expressions that we will pass to our dispatcher are next collected using `top-lvl-name->def`, a higher-level ViewForm function that takes a symbol denoting a top-level defined name and a context, and returns the whole `define` expression for the name. For `gunk:initial-value` and `gunk:set-particle-value!`, we pull out the define expression's body. For the selector `gunk:particle-value`, we do not, since one of our predicates tests for the `define-integrable` special form and expects to get the entire define expression. These viewcode expressions are combined into a list, then passed to our dispatcher. For aesthetics, we also write selectors that pull out each viewcode expression from the list passed to the dispatcher. These selectors are `vc-exps/init-val`, `vc-exps/mutator-val`, and `vc-exps/selector-def`. Finally, the control structure registers the coupling if the vforms completed successfully, and returns the created context. If the vforms were unsuccessful, the control structure signals the problem to the user.

Now we can create our coupler, `acs-flov-rule`, also shown in Figure 5-6. We use the generic `combine-vforms` function, the control structure and dispatcher we previously implemented, and we provide a new, more abstract explanation. `verbose-expl-combiner` is a ViewForm explanation combiner that prints its argument to the standard output in default mode, and invokes the explanations of its sub-vforms in verbose mode (controlled by the global boolean variable `*verbose*`). `fully-qualify` ensures that the filename passed to the control structure is a fully qualified filename. Figure 5-3 displays the particle abstraction code generated by the coupler. This code was generated after evaluating the expression `(do-vform acs-flov-rule #f '())`.

5.2.6 Invoking the View Invalidation/Recoupler

One of the simulator functions, `gunk:diffuse-neighbor!` is given below. This function is called many times within a simulator iteration, to diffuse a particle's values to its neighbors. Blair found that the MIT Scheme compiler stack allocated the values named by the `let*` form (i.e., `neighbor` and `exchange`). In doing so, the compiler boxed the values, thereby creating a pointer to a heap-allocated floating point number. This, unfortunately, leads to more boxing, floating point number garbage, and reduced performance.

```
(define-integrable (gunk:diffuse-neighbor! fix:i particle neighbors)
  (let* ((neighbor (gunk:get-neighbor neighbors fix:i))
         (exchange (flo:* *gunk:diffusion-rate*
                          (flo:- (gunk:particle-value particle)
                                 (gunk:particle-value neighbor)))))
    (cond ((not (gunk:particle-special-tag neighbor))
           ( gunk:set-particle-value!      neighbor
                                           (flo:+ (gunk:particle-value neighbor)
                                                  exchange))))
    (cond ((not (gunk:particle-special-tag particle))
           ( gunk:set-particle-value!      particle
                                           (flo:- (gunk:particle-value particle)
                                                  exchange))))))
```

The MIT Scheme compiler, however, will not box the intermediate float if `exchange` is desugared out of the `let*` expression as follows:

```
(define-integrable (gunk:diffuse-neighbor! fix:i particle neighbors)
  (let ((neighbor (gunk:get-neighbor neighbors fix:i)))
    ((lambda (exchange)
       (cond ((not (gunk:particle-special-tag neighbor))
              (gunk:set-particle-value! neighbor
                                        (flo:+ (gunk:particle-value neighbor)
                                               exchange))))
       (cond ((not (gunk:particle-special-tag particle))
              (gunk:set-particle-value! particle
                                        (flo:- (gunk:particle-value particle)
                                               exchange)))))
     (flo:* *gunk:diffusion-rate*
            (flo:- (gunk:particle-value particle)
                   (gunk:particle-value neighbor))))))
```

After making this change, the view invalidation/recoupler was invoked by evaluating the expression: `(vir!)`. ViewForm then recognized the change and re-invoked our coupler. The coupler revalidated the predicates and subsequently regenerated the same code as before (i.e., in Figure 5-3). This is what was expected, as the change above does not invalidate the preconditions.

5.2.7 Discussion

The most difficult and time-consuming aspect of `acs-flov-rule`'s construction was developing the preconditions and a safe, robust set of modifications. Preconditions are difficult because they must be general, correct, and must make sense in the presence of future code maintenance. Modifications are difficult because they must be correct and should have a limited scope. Once the preconditions and modifications were determined, expressing them was straightforward.

Various subtleties exist in the action templates for the simulator example. For instance, in `action/flov-init-val`, we do not `let`-capture the initial value, because the MIT Scheme compiler would otherwise box and heap-allocate the resulting float. By desugaring the implicit `let` into an applied `lambda`, the MIT Scheme compiler will avoid heap allocation. This kind of modification is

necessary whether or not view-based abstraction is used. In addition, the same code template binds and names the initial particle value even though this name is used only once. Rather than being a case of aesthetic design, the binding is necessary. We cannot allow the variable naming the flovec to be captured by the initial value expression. This problem can be otherwise addressed with hygienic macro[15] templates or in MIT Scheme by generating uninterned symbols in some circumstances.

While the flovec predicates are straightforward expressions of the preconditions, they might not have been under other circumstances. Both `predicate/returns-inexact` and `predicate/flov-mutator` depend on a data-flow analysis. These kinds of analyses can lose precision when values flow into and out of data structures, closures, or top-level variable names. While none of these situations happened in the simulator example, some contingencies for this situation were discussed. Nevertheless, when data flow fails to determine the desired properties, either a different set of preconditions must be used or user interaction must provide the missing information.

Implementation-specific language properties, such as floating point number identity, are difficult to manage because they may interfere with otherwise valid preconditions. Language implementations may also contain a variety of extensions. When these extensions significantly affect or alter the semantics of the original language, views that assume a strict adherence to the real language specification may not produce information consistent with a program's semantics. In these cases, new views must be written or acquired. The ability to decompose, recompose, and combine views is a form of incrementality that can attenuate the need to rewrite views for every language extension. Still, the need can remain.

5.2.8 Quantitative Measurements

The following tables contain the quantities outlined in the experimental methodology. They will be discussed later, along with the data collected from the other examples.

Simulator Example	Number of Constructs	Total Number of Lines	Average Number of Lines per Construct
Predicates	4	8	2
Actions	3	24	8
Dispatchers	1	9	9
Rules	1	21	21
New Views	0	0	0
Other	N/A	3	N/A
Total	9	65	N/A

Number of Lines the Action Modified	14
Number of Lines the Action Depended Upon	1
Number of Lines Analyzed	~1100 (code)
Time to Couple Code	8.7 seconds
Time to Recouple Code	9.2 seconds
Time to Compile Code	25.3 seconds

5.3 Pedigree Example

The pedigree example is Blair's[7] adaptation of a computational approach to determining conditional probabilities for pedigrees[83]. The purpose of such a computation is to determine the conditional probability of an individual having a particular genotype given a set of observed phenotypes in related individuals. [83] shows how such a conditional probability computation can be optimized using an algebraic formula factoring method based on a Bayes network model of a pedigree tree.

```
(define model-2+i:G_M (pedigree/unknown/one-of-N/even-odds 10)) ; Genotype: 1 / 10
(define model-2+i:G_F (pedigree/unknown/one-of-N/even-odds 10))
(define model-2+i:G_D (pedigree/unknown/one-of-N/even-odds 10))
(define model-2+i:G_P (pedigree/unknown/one-of-N/even-odds 10))
(define model-2+i:P_P (pedigree/unknown/one-of-N/int-boole  5)) ; Phenotype: 1 / 8

(define (P_pedigree)
  (p-sum G_D
         (lambda (g_d)
           (* (p-sum G_F
                     (lambda (g_f)
                       (* (P g_f '())
                          (P gP  '())
                          (p-sum P_P
                                 (lambda (p_p)
                                   (p-sum G_P
                                          (lambda (g_p)
                                            (* (P p_p `(,g_p))
                                               (p-sum G_M
                                                      (lambda (g_m)
                                                        (* (P g_p `(,gP ,g_m ,g_f))
                                                           (P g_d `(    ,g_m ,g_f))
                                                           (P PM  `(    ,g_m     ))
                                                           (P            g_m   '())
                                                           ))))))))))
                          (P PF `(,g_f)))))
              (P TD `(,g_d))
              (P PD `(,g_d))
              (P OM `(,PM ))))))
```

Figure 5-7 - Pedigree Computation Code

```
(define (pedigree/unknown/one-of-N/even-odds N) ; Flonum valued params
  (make-vector                                  N  (/ 1.0 N)))

(define (pedigree/unknown/one-of-N/int-boole N) ; Boolean valued params
  (make-initialized-vector                    N (lambda (i) (if (even? i) 0 1))))

(define (p-sum v proc)
  (do ((index (-1+ (vector-length v)) (-1+ index))
       (acc             0             (+ acc (proc (vector-ref v index)))))
      ((negative? index) acc)))

(define (P val givens)
  (let ((given-val (reduce * 1 givens)))
    (/ (* val given-val)
       given-val)))

(define (sc-* thunks)
  (do ((thunks-2-do thunks (cdr thunks-2-do))
       (acc           1    (* acc ((car thunks-2-do)))))
      ((or (null? thunks-2-do)           (zero? acc))       acc)))
```

Figure 5-8 - Pedigree Data Abstraction Implementation

As with the simulator example, Blair's objective with the pedigree example was to optimize the running time of the code. For this purpose, Blair developed an instance of the pedigree problem to experiment with. The code for this instance implements a fixed pedigree tree (but not a fixed set of observed phenotypes). The bulk of the pedigree computation for this instance is performed by the `P_pedigree` procedure in Figure 5-7 (part of a larger program developed by Blair). In this code, `P_pedigree` is used to compute a conditional probability based on various sets of phenotype observations, each represented by a set of variables such as the `model-2` variables given in Figure 5-7. Blair's code `set!`'s the free variables in `P_pedigree` to the values of these model variables before `P_pedigree` is invoked. For the pedigree example, we will write two rules and combine them into a coupler. The first rule is for short circuiting multiplication. This coupling was developed by Blair, based on an optimization suggested in [83], and will require a new view. The second rule in a function inlining. Each rule's purpose is described first below, before their implementations are presented.

5.3.1 Short-Circuiting Multiplication

For the `P_pedigree` code, one optimization discussed in [83] is short-circuiting the multiplication of a set of numbers. This means stopping immediately and returning zero if a multiplicand equal to zero is found during the multiplication of a set of numbers. Blair accomplished this by replacing `*` in `P_pedigree` with `sc-*`, his implementation of short-circuiting multiplication (given in Figure 5-8). In fact, the version of `P_pedigree` originally provided by Blair was the `sc-*`-based implementation given in Figure 5-9. To demonstrate how ViewForm can enhance code maintainability, we will write a rule for automatically replacing `*` with `sc-*` on the code in Figure 5-7.

```
(define (P_pedigree)
  (p-sum G_D
         (lambda (g_d)
           (sc-*
            `(,(lambda ()
                 (p-sum G_F
                        (lambda (g_f)
                         (sc-*
                          `(,(lambda () (P g_f '()))
                            ,(lambda () (P gP  '()))
                            ,(lambda ()
                               (p-sum
                                P_P
                                (lambda (p_p)
                                  (p-sum G_P
                                         (lambda (g_p)
                                           (sc-*
                                            `(,(lambda () (P p_p `(,g_p)))
                                              ,(lambda ()
                                                 (p-sum
                                                  G_M
                                                  (lambda (g_m)
                                                    (sc-*
                                                     `(,(lambda () (P g_p `(,gP ,g_m ,g_f)))
                                                       ,(lambda () (P g_d `(    ,g_m ,g_f)))
                                                       ,(lambda () (P PM  `(    ,g_m     )))
                                                       ,(lambda () (P            g_m   '()))
                                                       ))))))))))))))))
                            ,(lambda () (P PF `(,g_f)))))))))
            ,(lambda () (P TD `(,g_d)))
            ,(lambda () (P PD `(,g_d)))
            ,(lambda () (P OM `(,PM )))))))))
```

Figure 5-9 - Desired `P_pedigree`

This rule's goal is to produce the code written by Blair in Figure 5-9 from the code in Figure 5-7. While this rule does not produce a bona fide implementation coupling (for this case), it does provide at least three distinct benefits. The most obvious is readability. The code in Figure 5-7 is no doubt more readable than the code in Figure 5-9, and is therefore easier to maintain. The second benefit is that if a maintenance programmer modifies the `P_pedigree` implementation in Figure 5-7, our rule will replace all instances of * with `sc-*` without any further effort by the programmer. This frees the maintenance programmer from having to deal with short-circuit multiplication when modifying the code, thus enhancing maintainability by making the programmer's task less complex. The third benefit is robustness. If a maintenance programmer replaces any * expressions in Figure 5-7 with some other expression that reduces to *, our rule would replace those expressions, too. In this case, the rule may actually produce an imple-

mentation coupling.[§§§§] To carry out its tasks, our rule requires a new view; the comb view. While simple, this view nevertheless demonstrates a view implementation for code I did not write for a modification I did not suggest or design.

5.3.2 Function Inlining

The second rule implemented for the pedigree example is a function inliner. The functions to be inlined are `p-sum` and `p`, whose implementation is given in Figure 5-8. These functions present an interface used by `P_pedigree` to compute the desired conditional probabilities. Both `p-sum` and `p` are in the same module as `pedigree/unknown/one-of-N/even-odds` and `pedigree/unknown/one-of-N/int-boole`, also given in Figure 5-8. The reason that inlining `p-sum` and `p` into `P_pedigree` is an abstraction violation is that the `p-sum` and `p`'s implementations depend on the functions `pedigree/unknown/one-of-N/even-odds` and `pedigree/unknown/one-of-N/int-boole` using vector representations. After inlining `p-sum` and `p` into `P_pedigree`, an invalidating implementation of `pedigree/unknown/one-of-N/even-odds`, for example, would use lists instead of vectors. Such a representation shift would invalidate the implementation-coupled `P_pedigree`.

The goal for the inlining aspect of the pedigree example is to implement a rule expressing such an implementation coupling using ViewForm.

5.3.3 Implementing the Short-Circuiting Multiplication Rule

The goal of the short-circuiting multiplication rule is to replace each operator that calls `*` with `sc-*`, and to modify the arguments to be thunks (i.e., a `lambda` of no arguments), as expected by `sc-*`. This is implemented in two steps. The first is to go through the implementation-coupling steps (I-VI) and the view-based abstraction methodology (i-vi) to implement a rule. This rule will depend on a view not provided by default. The second step is to implement that view. We begin by implementing the rule.

5.3.3.1 Modules/Contexts

The pedigree example has two relevant modules. One contains the phenotype data abstraction and computation functions. The relevant functions in this module (which corresponds to M_2 in Figure 1-1) are given in Figure 5-8. We will call this module M_g. The second module contains code for computing the conditional probability. We will call this module M_p. The relevant code for this module is given in Figure 5-7. M_p and M_g will be the basis for the contexts we create later. These contexts will also be used by the function inlining rule described earlier.

[§§§§] For instance, if `*` was replaced by the variable `mul`, and `mul` was defined in a different module to be `*`, replacing `mul` by `sc-*` would produce an implementation coupling.

5.3.3.2 Preconditions/Predicates

The preconditions needed to validate the substitution of `sc-*` for * in a given viewcode expression are:

- The viewcode expression is a combination
- The combination's operator always evaluates to *'s value

The first precondition is handled by the comb view we will write later. This view will provide our predicate with viewcode expressions that are solely combinations. The second precondition is implemented by `predicate/sc-*` below. This predicate determines whether a combination's operator value is always the primitive *'s value. This satisfies the the second precondition above.

This predicate itself takes a single viewcode expression (i.e., the `car` of `vc-exps`), assumes it is a combination, gets the expression in the combination's operator position, and collects all of the expressions producing the operator's procedure values via the liar view function `exp->all-procs`. This collected procedure producer list contains the information we need to validate the precondition. The predicate goes through each procedure producer expression in this list and checks whether the expression is the predefined procedure *. If so, the predicate returns a true value.

```
(define predicate/sc-*
  (make-explained-predicate
    (lambda (vform context vc-exps)
     (let ((result (map (lambda (proc) (eq? (is-predefined proc) '*))
                        (exp->all-procs (combination/proc (car vc-exps)) context))))
       (for-all? result identity-procedure)))
   "ensuring proc calls only *"))
```

This predicate could easily be parameterized on the symbol denoting the predefined procedure (e.g., *). Such a parameterized predicate would be a valuable addition to a ViewForm library.

5.3.3.3 Views

`predicate/sc-*` needs one view, the liar view (because of `exp->all-procs`). It also implicitly needs the comb view, which collects the combinations contained in a group of viewcode expressions. The comb view is defined later, after we have finished implementing the `sc-*` rule.

5.3.3.4 Modifications/Actions

The goal of the short-circuit multiplication example is to change all combinations with operators calling * into combinations that call `sc-*`. Since `sc-*` takes a list of thunks as arguments instead of a set of numbers, our modification must also create thunk expressions out of the number-generating argument expressions to *, and collect them into a list. This modification is best illustrated by Figure 5-9.

The action below implements this modification. It first replaces the operator with `sc-*`. Next, it replaces each argument with the (unquoted) thunk `(lambda () <argument>)`. Then, it replaces the arguments with an expression creating a list of those (quasiquoted) arguments. Quasiquoting and unquoting were used to make the results consistent with Blair's code.

```
(define action/sc-*
  (make-simple-action (lambda (vform context vc-exp)
                        (let ((proc (combination/proc vc-exp))
                              (args (combination/args vc-exp)))
                          (replace proc 'sc-*)
                          (for-each (lambda (arg)
                                      (replace arg (list 'unquote `(lambda () ,arg))))
                                    args)
                          (replace args (list (list 'quasiquote args)))
                          #t))
                      "replacing * with sc-*"))
```

5.3.3.5 A Dispatcher

As is usually the case, the dispatcher combining the predicate and action is straightforward to implement. `make-simple-dispatcher` uses a simple explanation combiner that returns an explanation that explains both the predicate and action when invoked.

```
(define dispatcher/sc-*  (make-simple-dispatcher predicate/sc-* action/sc-*))
```

5.3.3.6 A Rule

The next step is to write a rule that expresses a short-circuit multiplication substitution. This rule collects the viewcode expressions to be passed to our dispatcher, invokes any necessary views on those expressions, and invokes the dispatcher. These steps are similar to what a coupler does, except that the rule does not create contexts or register a coupling with the view invalidation/recoupler. That will be our coupler's job.

To write this rule, we begin with the control structure constructor below. It takes one parameter, `repl-sc-*-symbols`, which is a list of symbols denoting a set of top-level names defined in the context's expressions (i.e., `P_pedigree`) whose combinations are to be examined. `top-lvl-name->def` is then used to map the symbols into the corresponding viewcode definition expressions (e.g., `(define (P_pedigree) …)`). If a definition expression is found for each symbol, we proceed to compute the comb view on all the definitions. The comb view recursively descends each definition expression, collecting combinations as it encounters them. In the code below, this set of expressions examined by the comb view is `replace-defs`. The vforms are then each invoked on each combination found by the comb view. If any substitution was by a single vform, a true value is returned indicating a coupling has been made. Our coupler will use this return value to register the coupling with the view invalidation/recoupler.

```
(define (make-replace-sc-*-control-structure repl-sc-*-symbols)
  (lambda (vforms)
    (lambda (vform context vc-exps)
      (let ((replace-defs (map (lambda (binding) (top-lvl-name->def binding context))
                               repl-sc-*-symbols)))
        (if (for-all? replace-defs identity-procedure)
            (let ((found-one? #f))
              (ensure-comb-view-computed! context replace-defs)
              (for-each (lambda (vform)
                          (for-each (lambda (vc-exp)
                                      (if (do-vform vform context (list vc-exp))
                                          (set! found-one? #t)))
                                    (context/comb-walker context)))
                        vforms)
              found-one?))))))
```

The short-circuit multiplication rule can now be created using this control structure and our previously defined dispatcher. This code is given below, and will be used in our pedigree coupler. `verbose-expl-combiner` was described earlier, as part of the simulator example.

```
(define (make-replace-sc-*-rule repl-sc-*-symbols)
  (combine-vforms (make-replace-sc-*-control-structure repl-sc-*-symbols)
                  (verbose-expl-combiner "replacing * with sc-*")
                  (list dispatcher/sc-*)))
```

5.3.4 Comb View

We now turn to the implementation of the comb view. The purpose of the comb view is to collect the set of all combinations lexically dominated by a set of given expressions within a context. This view is implemented in much the same way as the `*lambda-view*` from Section 4.6.4, via the following five steps:

1. Implement the merging and copying procedures, as well as the view initialization value
2. Implement the view dispatchers
3. Combine the dispatchers into a view
4. Register the view
5. Provide a way to invoke the view

For the first step, we must decide upon a representation for the view information. For the comb view, a simple list will suffice. The corresponding merge, destructive merge, copying, and initial value are therefore simple list operations:

```
(define (comb-walker-merge  view1 view2) (append  view1 (list-copy view2)))
(define (comb-walker-merge! view1 view2) (append! view1            view2))
(define  comb-walker-copy   list-copy)
(define  comb-walker-init   '())
```

The next step is to implement the view dispatchers. For the comb view, we need only one since we can rely on the walker view to provide a recursive decent. This dispatcher's predicate deter-

mines whether an expression is a combination, and the action will store the combination into the comb view's data structure, in the appropriate context. The action, dispatcher, and predicate are given below. The action uses the helper functions `set-context/comb-walker!` and `context/comb-walker` to set and retrieve the comb view list, respectively. The view dispatcher, `walker/comb-finder`, uses `make-predicate/exp-type` to create a view predicate that looks for combinations (this predicate constructor was also used in the simulator example).

```
(define (context/comb-walker context)
  (or (retrieve-view *comb-view* context) comb-walker-init))

(define (set-context/comb-walker! context view) (set-view! *comb-view* context view))

(define action/store-comb
  (make-simple-action (lambda (vform context vc-exp)
                        (set-context/comb-walker! context
                                                  (cons vc-exp (context/comb-walker context))))
                      "storing a combination"))

(define walker/comb-finder
  (make-dispatcher (make-predicate/exp-type type/combination? "combinations")
                   action/store-comb
                   (verbose-expl-combiner "for collecting combinations")))
```

The third step is to create the view itself. We name this view `*comb-view*`. The view's control structure needs to clear the old view information data structure (so that repeated invocations of the view do not pile on extra combinations). It then must invoke the view dispatcher on each viewcode expression passed to it. To recursively walk the viewcode subexpressions, the view dispatcher is combined with the `*walker-vform*`.

```
(define (clear-comb-view! context) (set-context/comb-walker! context comb-walker-init))
(define *comb-view*
  (make-view (lambda (vforms)
               (let ((combined-vform (apply combine-vforms-do-all vforms)))
                 (lambda (vform context vc-exps)
                   (clear-comb-view! context)
                   (for-each (lambda (vc-exp) (do-vform combined-vform context (list vc-exp)))
                             vc-exps))))
             (verbose-expl-combiner "comb-walker combination collector")
             (list walker/comb-finder *walker-vform*)))
```

The fourth step is registering the view. Since we have defined all the components needed to do this, we simply call `register-view!` with these components.

```
(register-view! *comb-view* comb-walker-merge comb-walker-merge!
                            comb-walker-copy  comb-walker-init)
```

The final step is to provide a way to invoke the view. The procedure below invokes the view only if there is no comb view information available. To recompute the view after it was previously invoked, a programmer can call `clear-comb-view!` before calling `ensure-comb-view-computed!`.

```
(define (ensure-comb-view-computed! context vc-exps)
  (if (null? (context/comb-walker context))
      (do-vform *comb-view* context vc-exps)))
```

5.3.5 Function Inlining

Now that we have implemented the first rule for the pedigree example, we can continue to the second rule. Our goal is to produce a rule that, given a symbol denoting a top-level defined variable name, ensures that it is inlined everywhere it appears in an operator position. The MIT Scheme syntaxer can produce such an inlining if the variable's definition is modified as follows:

```
(define            <var> (lambda <formals> <body>)) ➔
(define-integrable <var> (lambda <formals> <body>))
```

When the MIT Scheme syntaxer sees a `define-integrable` expression, it replaces any operator-position references to `<var>` with `<body>`. [*****] Any references to the parameters in `<formals>` are also replaced with the corresponding argument expressions. For example, given:

```
(define count 1)
(define-integrable (foo bar) (bar (bar 1)))
(foo (begin (set! count (1+ count)) +))
```

the MIT Scheme syntaxer produces the following transformation:

```
(begin (define count 1)
       (define (foo bar) (bar (bar 1)))
       ((begin (set! count (1+ count)) +) ((begin (set! count (1+ count)) +) 1)))
```

The problem with this kind of inlining, as is exemplified by the code above, is that any side effects within the arguments (i.e., `bar`) to the inlined function may end up being performed more than once. Changing a definition from `define` to `define-integrable` is therefore not necessarily semantics preserving. To get around this problem, we use the following modification instead:

```
(define <var>
  (lambda (<formal1> … <formaln>)
    <body>)) ➔
(define-integrable <var>
  (lambda (<formal1> … <formaln>)
    ((lambda (<formal1> … <formaln>) <body>))
       <formal1> … <formaln>))
```

This transformation is a desugared `let`-binding of the formals. Only one instance of the formals will thus be replaced by the inlining (the formals passed to the invoked `lambda`). For example, given the code:

[*****] The inlining is performed in the same file, unless some other file has an `integrate-external` declaration. The MIT Scheme User's Manual contains more details on `define-integrable` and `integrate-external`.

```
(define count 1)
(define-integrable (foo bar)
  ((lambda (bar) (bar (bar 1)))
   bar))
(foo (begin (set! count (1+ count)) +))
```

The MIT Scheme syntaxer now produces the code below. This code performs the side effect to `count` only once, unlike what happened in the earlier example. Another important consequence of this modification is that `<body>` can no longer capture variables in any substituted expression.

```
(begin (define count 1)
       (define (foo bar) (let ((bar bar)) (bar (bar 1))))
       (let ((bar (begin (set! count (1+ count)) +)))
         (bar (bar 1))))
```

Our goal is to write a rule to perform function inlining this way on `p-sum` and `p`. Performance measurements indicate this kind of simple inlining can produce a 10% speedup in `P_pedigree`. We will implement this rule using the view-based abstraction methodology.

5.3.5.1 Modules/Contexts

The function inlining rule uses the modules M_g and M_p from the short-circuit multiplication rule. Constructing the contexts from these modules will be implemented later, in the coupler.

5.3.5.2 Preconditions/Predicates

The procedures `p-sum` and `p` can be inlined under the three preconditions given below. These preconditions assume our actions will produce the inlining modifications outlined above.

- `p-sum` and `p` are not side effected
- `p-sum` and `p` do not reference any free variables "captured" by `P_pedigree`
- `p-sum` and `p` do not maintain local state shared among invocations

The first precondition ensures that `p-sum` and `p` are not redefined elsewhere. This precondition can be tested using a predicate similar to one defined in Chapter 4. The predicate, given below, takes a variable binding (the `car` of `vc-exps`) and ensures that it is not mutated. It does this using the alpha view function `var-binding->mutations`, which returns a list of a variable binding's mutation sites. If this list is empty, the variable is not mutated in the context.

```
(define predicate/no-mutations
  (make-explained-predicate (lambda (vform context vc-exps)
                              (let ((binding (car vc-exps)))
                                (null? (var-binding->mutations binding context))))
   "a non-mutated variable"))
```

The second precondition ensures that `p-sum` and `p` do not contain any variables that could be captured by `P_pedigree`. If `p-sum` were to reference a free variable `foo`, for example, and

P_pedigree were to have a formal foo, inlining p-sum's body would cause p-sum's free variable reference to foo to refer to P_pedigree's formal foo. We can implement this precondition as a set of two predicates. The first is predicate/usual-integrations, defined for the simulator example. This ensures that the primitive variables in p-sum and p have known values. The second predicate ensures that no free variables exist in p-sum and p. It uses the alpha view function top-lvl-exp->free-vars, which returns a list of a top-level expression's free variables. If this list is empty, then p-sum and p contain no free variables. These predicates could be made more weak and less fragile, since a free variable is a problem only if it is bound inside P_pedigree or if the third precondition is violated. This lower degree of fragility could be achieved by performing an alpha renaming on all free variables in p-sum and p. Since the predicate is more illustrative for the purposes of this dissertation, the predicate route is pursued:

```
(define predicate/no-free-vars
  (make-explained-predicate
   (lambda (vform context vc-exps)
     (let ((vc-exp (first vc-exps)))
       (null? (top-lvl-exp->free-vars (exp->top-level vc-exp) context))))
   "no free variables"))
```

The third precondition means that p-sum and p cannot have definitions such as the following:

```
(define p-sum
  (let ((state-shared-among-invocations (make-eq-hash-table)))
    (lambda (v proc)
      …)))
```

This kind of shared state would no longer be shared if p-sum were inlined. To validate the third precondition, we write a predicate that checks whether p-sum's value in its defining expression is generated directly by a lambda. In the shared-state p-sum code above, for example, p-sum's value is generated by a let expression. The predicate below takes a variable binding expression (as the car of vc-exps), up links two levels to its defining expression, extracts the expression generating the variable's value, and tests whether it is a lambda expression. Thus, the predicate below returns true only if the defining expression has the form (define <var> (lambda … …)). While this predicate is more fragile than the precondition, it is simpler than a full semantic analysis on any let or other expressions generating <var>'s value (which nevertheless, could alternatively be expressed).

```
(define predicate/no-local-storage
  (make-explained-predicate (lambda (vform context vc-exps)
                              (let ((binding (car vc-exps)))
                                (and binding
                                  (type/lambda? (define-var/value (up-link-n binding 2))))))
   "an expression having no local storage"))
```

5.3.5.3 A Modification and Action

The previously discussed modification is implemented with the action given below. This action is passed a set of viewcode expressions, and performs the desired modification on each one. Each viewcode expression is a variable binding site previously validated by our predicates. The modification is implemented as a code template that recombines the original definition's `lambda` formals and body.

```
(define action/make-define-integrable
  (make-explained-action
   (lambda (vform context vc-exps)
     (for-each (lambda (vc-exp)
                 (let* ((define-exp (up-link-n vc-exp 2))
                        (formals    (lambda/formals (define-var/value define-exp))))
                   (replace define-exp `(define-integrable ,(define-var/name define-exp)
                                          (lambda ,formals
                                            ((lambda ,formals
                                               ,(lambda/body (define-var/value define-exp)))
                                             ,@formals))))))
               vc-exps)
     #t)
   "function inlining"))
```

5.3.5.4 A Dispatcher

The dispatcher is created using the four predicates and the action. The predicates are combined into a compound predicate that returns true only if they all return true.

```
(define dispatcher/def-integrable
  (make-simple-dispatcher (combine-vforms-conjunctive predicate/no-mutations
                                                      predicate/no-free-vars
                                                      predicate/usual-integrations
                                                      predicate/no-local-storage)
                          action/make-define-integrable))
```

5.3.5.5 A Rule

The goal of our inlining rule is to perform the inlining modification on a set of `define` expressions. To make this rule simpler to invoke, we parameterize its control structure on a list of symbols corresponding to the top-level names defined in the context of interest (i.e., a list of `p-sum` and `p` in the pedigree example). The control structure for this rule will then ensure each symbol is, in fact, defined in the context. The alpha view function `top-level-defined?` returns a viewcode variable binding if one exists. These binding expressions are used as the viewcode expression list passed to the combined vforms. This control structure is given below.

Using this control structure constructor, we build a rule constructor parameterized on the binding symbols. This rule constructor will be used to implement the pedigree example coupler.

```
(define (make-define-integrable-control-structure binding-symbols)
  (lambda (vforms)
    (lambda (vform context vc-exps)
      (let ((bindings (map (lambda (binding) (top-level-defined? binding context))
                           binding-symbols)))
        (and (for-all? bindings identity-procedure)
             (for-all? vforms
               (lambda (vform) (do-vform vform context bindings))))))))

(define (make-define-integrable-rule binding-symbols)
  (combine-vforms (make-define-integrable-control-structure binding-symbols)
                  (verbose-expl-combiner "integrating define expressions")
                  (list dispatcher/def-integrable)))
```

5.3.6 Combining the Rules into a Coupler

The coupler for the pedigree example must implement both the short-circuit multiplication substitution and the function inlining. This is implemented by combining the short-circuit rule and the inlining rule into a coupler. The rules are supported and mediated by the coupler's control structure. As with other coupler control structures, this one is responsible for creating the contexts, ensuring the proper views are computed, invoking the appropriate rules or dispatchers, and registering the coupling if necessary. For our coupler, the context will be created from a set of passed-in filenames and the views to be computed are the alpha and liar views. This control structure, given below, follows the same basic structure as all previously illustrated coupler control structures.

```
(define (make-pedigree-control-structure file1 . rest-files)
  (lambda (vforms)
    (lambda (vform context vc-exps)
      (let ((context (or context (apply make-context-from-files (cons file1 rest-files)))))
        (ensure-liar-view-computed!  context)
        (ensure-alpha-view-computed! context)
        (if (for-all? vforms
              (lambda (vform)
                (do-vform vform context vc-exps)))
            (register-coupling! context vform))
        context))))
```

The coupler can now be defined quite easily. It combines the short-circuit and inlining rules, and passes the appropriate files to the control structure constructor.

```
(define (make-pedigree-rule binding-symbols repl-sc-*-symbols)
  (combine-vforms (make-pedigree-control-structure
                   (fully-qualify *pedigree-dir* "pedigree-data-abstraction")
                   (fully-qualify *pedigree-dir* "pedigree"))
                  (verbose-expl-combiner "integrating definitions")
                  (list (make-define-integrable-rule binding-symbols)
                        (make-replace-sc-*-rule      repl-sc-*-symbols))))
(define coupler/pedigree-rule (make-pedigree-rule '(p-sum p) '(p_pedigree)))
```

5.3.7 ViewForm Generated Code

The pedigree coupler is invoked by evaluating the following expression: `(do-vform coupler/pedigree-rule #f '())`. The output code is given below. The coupler generates a context containing the definitions below for `p-sum`, `p`, and `P_pedigree`. This generated code can then be syntaxed by the MIT Scheme syntaxer to generate a version of `P_pedigree` with the inlined `p-sum` and `p`. This syntaxed version produces the same answer as the original `P_pedigree` for the `model-2` set of phenotype observations (probability = 0.015625).

```
(define-integrable p-sum (lambda (v proc)
                           ((lambda (v proc)
                              (do ((index (-1+ (vector-length v)) (-1+ index))
                                   (acc 0 (+ acc (proc (vector-ref v index)))))
                                  ((negative? index) acc)))
                            v proc)))

(define-integrable p (lambda (val givens)
                       ((lambda (val givens)
                          (let ((given-val (reduce * 1 givens)))
                            (/ (* val given-val) given-val)))
                        val givens)))

(define
  p_pedigree
  (lambda ()
    (p-sum
     g_d
     (lambda (g_d)
       (sc-*
        (quasiquote
         ((unquote
           (lambda ()
             (p-sum
              g_f
              (lambda (g_f)
                (sc-*
                 (quasiquote
                  ((unquote (lambda () (p g_f (quote ()))))
                   (unquote (lambda () (p gp (quote ()))))
                   (unquote (lambda ()
                              (p-sum
                               p_p
                               (lambda (p_p)
                                 (p-sum
                                  g_p
                                  (lambda (g_p)
                                    (sc-*
                                     (quasiquote
                                      ((unquote (lambda ()
                                                  (p p_p (quasiquote ((unquote g_p))))))
                                       (unquote (lambda ()
                                                  (p-sum
                                                   g_m
                                                   (lambda (g_m)
```

```
                                                (sc-*
                                                 (quasiquote
                                                  ((unquote
                                                    (lambda ()
                                                      (p g_p (quasiquote
                                                              ((unquote gp)
                                                               (unquote g_m)
                                                               (unquote g_f))))))
                                                   (unquote
                                                    (lambda ()
                                                      (p g_d (quasiquote
                                                              ((unquote g_m)
                                                               (unquote g_f))))))
                                                   (unquote
                                                    (lambda ()
                                                      (p pm (quasiquote
                                                             ((unquote g_m))))))
                                                   (unquote
                                                    (lambda ()
                                                      (p g_m (quote ((((((((((((((((((((((((
          (unquote (lambda () (p pf (quasiquote ((unquote g_f))))))))))))))
     (unquote (lambda () (p td (quasiquote ((unquote g_d))))))
     (unquote (lambda () (p pd (quasiquote ((unquote g_d))))))
     (unquote (lambda () (p om (quasiquote ((unquote pm)))))))))))))
```

5.3.8 Modifying the Pedigree Code

One possible modification to the pedigree code is to change the implementations of `p-sum` and `p` to use non-generic floating-point arithmetic as follows:

```
(define (p-sum v proc)
  (do ((index (-1+ (vector-length v)) (-1+ index))
       (acc             0.           (flo:+ acc (proc (vector-ref v index)))))
      ((negative? index) acc)))

(define (P val givens)
  (let ((given-val (reduce * 1. givens)))
    (flo:/ (flo:* val given-val)
       given-val)))
```

This change invalidates the previously generated implementation coupling. After this modification was made, the view invalidation/recoupler was invoked. The change was recognized, and ViewForm then regenerated the same `P_pedigree` code as before, but with versions of `p-sum` and `p` reflecting the modifications above.

5.3.9 Quantitative Measurements

The following tables contain the quantities outlined in the experimental methodology. They will be discussed later, along with the data collected from the other examples.

Pedigree Example	Number of Constructs	Total Number of Lines	Average Number of Lines per Construct
Predicates	5	24	4.8
Actions	2	25	12.5
Dispatchers	2	7	3.5
Rules	3	49	16.3
New Views	1	31	31
Other	N/A	1	N/A
Total	13	137	N/A

Number of Lines the Action Modified	31
Number of Other Lines the Action Depended Upon	0
Number of Lines Analyzed	167
Time to Couple Code	1.27 seconds
Time to Recouple Code	1.25 seconds
Time to Compile Code	4.95 seconds

5.4 ViewForm Example

ViewForm's performance can benefit from implementation coupling. Via Blair's profiler[7], I found one dramatic performance bottleneck in the liar data-flow analysis implementation. The data-flow analysis is implemented using MIT Scheme's bitvectors to represent an initial data-flow graph. This graph is then traversed using a hand-optimized version of the Floyd-Warshall [31] algorithm. The bottleneck was in the procedure `propagate-nodes!`, shown below.

```
(define (propagate-nodes! tc-matrix)
  (let* ((size    (vector-length tc-matrix))
         (tc-copy (make-initialized-vector
                    size
                   (lambda (idx)
                     (let ((result (bit-string-allocate size)))
                       (bit-string-move! result (vector-ref tc-matrix idx))
                       result)))))
    (let loop-k ((k          0)
                 (up-to-date tc-matrix)
                 (work-copy  tc-copy))
      (let loop-rows ((row 0))
        (cond ((fix:= k   size)
               (if (not (eq? up-to-date tc-matrix))
                   (let loop ((index 0))
                     (cond ((fix:= index size))
                           (else (vector-set! tc-matrix index (vector-ref up-to-date index))
                                 (loop (fix:1+ index)))))))
              ((fix:= row size) (loop-k (fix:1+ k)
                                        work-copy
                                        up-to-date))
              (else
               (let ((row-bit-string (vector-ref up-to-date row)))
                 ;; *** Want to replace bit-string-ref with new-bit-string-ref ***
                 (if (bit-string-ref row-bit-string k)
                     (bit-string-or! row-bit-string
                                     (vector-ref (or (and (fix:< k row) work-copy)
                                                     up-to-date)
                                                 k)))
                 (vector-set! up-to-date row (vector-ref work-copy row))
                 (vector-set! work-copy  row row-bit-string)
                 (loop-rows (fix:1+ row)))))))))
```

Blair's profiler showed that a significant amount of time was being spent in the Scheme primitive `bit-string-ref`. This primitive takes a bit string and an index, and returns `#t` if the indexed bit is set and `#f` otherwise. At my request, Stephen Adams wrote an Intel x86 specific version of `bit-string-ref` that eliminated this bottleneck. This code is given below.

```
(define-macro (deflap name . lap)
  `(define ,name
     (scode-eval
      ',((access lap->code (->environment '(compiler top-level)))
         name
         lap)
      system-global-environment)))
```

```
(define new-bit-string-ref
  (let ()
    (deflap new-bit-string-ref
      (entry-point new-bit-string-ref)
      (Scheme-object CONSTANT-0 #F)
      (Scheme-object CONSTANT-1 0)
      (equate new-bit-string-ref new-bitstring-ref-0)
      (word u #x303)
      (block-offset new-bitstring-ref-0)
      (LABEL new-bitstring-ref-0)
      (mov w (r 0) (@ro b 4 0))         ; eax <- bs
      (mov w (r 3) (@ro b 4 4))         ; ebx <- I
      (and w (r 0) (r 5))               ; remove tag
      (and w (r 3) (r 5))               ; remove tag
      (add w (r 4) (& 8))               ; pop bs and I
      (and w (@r 4) (r 5))              ; unmask return address
      (mov b (r 1) (& 31))              ; cl <- i & 31
      (and b (r 1) (r 3))
      (shr w (r 3) (& 5))               ; bs >> 5
      (mov w (r 0) (@roi b 0 8 3 4))    ; eax <- bs[i>>5]
      (shr w (r 0) (r 1))               ; shift selected bit to LSB
      (test w (r 0) (& 1))
      (jz b (@pcr is-false))
      (mov w (@ro b 6 8) (& #x20000000)) ; return #T
      (ret)
      (label is-false)
      (mov w (@ro b 6 8) (& 0))         ; return #F
      (ret)
      )
    new-bit-string-ref))
```

This lap code above defines a new primitive called `new-bit-string-ref`. To use this new primitive, references to `bit-string-ref` must be renamed to `new-bit-string-ref`. In addition, the lap code above must be included in the same file as any references to `new-bit-string-ref`. Our goal for the ViewForm example is to replace all references in `propagate-nodes!` to `bit-string-ref` with `new-bit-string-ref`, and to add the lap code. This goal poses two problems.

The first problem is that the new primitive does no type checking. Passing `new-bit-string-ref` something other than a bit string, for example, could cause the Scheme run time to fail catastrophically. Before inserting references to `new-bit-string-ref`, its potential arguments must therefore be examined and deemed safe with absolute certainty. A careful visual inspection of the `propagate-nodes!`, for example, shows a possible danger. `row-bit-string`, the variable whose value is supposed to be a bit string, has a value passed in externally through `tc-matrix` (which is `propagate-nodes!`'s formal parameter). If a module invokes `propagate-nodes!` on a vector of something other than bit strings, `new-bit-string-ref` would have a problem. This situation can happen as a result of a bug in `propagate-nodes!`, a bug in a call to `propagate-nodes!`, or maintenance programmer modifications to invokers of `propagate-nodes!`. Our desired modification therefore depends on the implementation of modules outside of the one for `propagate-nodes!`.

The second problem is the specificity of the `new-bit-string-ref` implementation. It only works on MIT Scheme running under an x86-compatible processor. This creates an implementation coupling between `propagate-nodes!` and the underlying processor implementation. If ViewForm is re-compiled to a different, incompatible processor, `new-bit-string-ref` should not be included. These two problems must be addressed in the implementation coupling code we will write below. As before, we follow the six-step view-based abstraction methodology.

5.4.1 Modules/Contexts

ViewForm consists of 28 files, each corresponding to a module. `propagate-nodes!` is in one of these modules. While only five other modules are allowed to call `propagate-nodes!`, we will assume that it might be called from any of the other modules. This is done to provide a better perspective on ViewForm's performance and ability to scale. Contexts for these 28 modules will be created and merged by the coupler we will write.

5.4.2 Preconditions/Predicates

As previously discussed, before `new-bit-string-ref` can be substituted, its two argument types must be validated to be a bit string and a positive integer, respectively.[†††††] In addition, the computing platform's processor must be x86 compatible. These preconditions are summarized as follows:

- Candidate expression for replacement must evaluate to `bit-string-ref`'s value
- The first argument of every combination whose operator is `bit-string-ref`'s value must be a bitstring
- The second argument of every combination whose operator is `bit-string-ref`'s value must be a non-negative integer
- The computing platform's processor must be x86 compatible

The first precondition will be managed by the coupler, which will extract the appropriate candidate expressions from the contexts it creates. The next two preconditions could be handled the same way as in the simulator example, with a simple combination of three predicates created from `make-predicate/check-ret-type` and `make-predicate/check-proc-params`. Instead, we take a less fragile route; one that involves user interaction. Our goal is to write a predicate constructor like `make-predicate/check-ret-type`. Our predicate constructor will take a type, and return a predicate that ensures that all viewcode expressions passed to it will return values consistent with that type. The difference is what happens when the precondition is not validated. Our new predi-

[†††††] We do not perform a range check on the index for this example, although in practice it should also be done.

cates should present any cases that would otherwise invalidate the precondition to the user, and ask the user to validate or invalidate each case. This is implemented with the predicate constructor code below.

```
(define (make-user-predicate/type-consistent type)
  (make-explained-predicate
   (lambda (vform context vc-exps)
     (for-all? vc-exps
       (lambda (exp)
         (for-all? (list-transform-negative (consumer->primitive-producers exp context)
                     (lambda (producer)
                       (for-all? (exp->return-types producer context)
                         (lambda (ret-type) (type-consistent? type ret-type)))))
           (lambda (bad-type-exp)
             (format #t "~%~%The expression: [~S] ~S ~%     returning types: ~S ~
                         ~%     is inconsistent with the required type: ~S ~
                         ~%     for the expression to be transformed: ~S"
                     (hash bad-type-exp) bad-type-exp (exp->return-types bad-type-exp context)
                     type exp)
             (y-or-n-user-response))))))
   "user-interactive type consistency checking"))
```

This code has two major parts. The first collects a list of a viewcode expression's primitive producers (e.g., expressions creating bit strings). It filters this list for any primitive producers that do not produce values consistent with the passed-in type (e.g., vector-creating expressions). Normally, the presence of any viewcode expressions in this list would invalidate the precondition. Our predicate, however, presents each of these viewcode expressions to the user. This is the second major part of the predicate. For each potentially invalidating viewcode type, our predicate prints a message out to the user, then asks for a yes or no response. This predicate constructor will be used to construct a predicate that checks for bit string-returning expressions and a predicate that checks for non-negative-integer returning expressions.

The final precondition can be tested using the platform view, to be defined later. The platform view maps a context's expressions to the computing platform they are or will be run on. One of the functions exported by this view is `x86-platform?`, which takes a context and returns a true value if the context's expressions are or will be run on an x86. A predicate implementing the final precondition is given below.

```
(define predicate/test-os
  (make-explained-predicate (lambda (vform context vc-exps)
                              (x86-platform? context))
                            "verifying x86 architecture"))
```

5.4.3 Views

The predicates above use three views. The first is the liar view (e.g., `exp->return-types`). The second, the alpha view, is implicit because it will be used by the coupler to look up all refer-

ences to the variable `bit-string-ref` to ensure the first precondition. The third required view is the platform view, which will be implemented later. Our coupler must ensure these views have been computed prior to the invocation of these predicates.

5.4.4 Modifications/Actions

We need two modifications for the ViewForm example. The first is to replace `bit-string-ref` by `new-bit-string-ref`. The second is to insert the lap code. The former is performed by `action/vf-repl-bitstring-ref` below. It takes a viewcode list of combinations (like our predicates above) and replaces each operator expression with `new-bit-string-ref`. The second modification is implemented by the action constructor below, `make-action/insert-code`. It takes an expression list and an explanation string, and returns an action that inserts the expression list expressions after the last expression in the given context. The action `action/vf-insert-lap`, also given below, inserts the lap code for the ViewForm example into the given context.

```
(define action/vf-repl-bitstring-ref
  (make-explained-action
   (lambda (vform context vc-exps)
     (for-each (lambda (vc-exp) (replace (combination/proc vc-exp) 'new-bit-string-ref))
               vc-exps)
     #t)
   "replacing exps with 'new-bit-string-ref"))

(define (make-action/insert-code code-list expl-string)
  (make-explained-action
   (lambda (vform context vc-exps)
     (let ((last-exp (car (last-pair (context/exps context)))))
       (for-each (lambda (code) (insert-after-in-context! context code last-exp))
                 code-list)
       #t))
   expl-string))

(define action/vf-insert-lap
  (make-action/insert-code
   (list '(define new-bit-string-ref
            (let ()
              (deflap new-bit-string-ref
                (entry-point new-bit-string-ref)
                (Scheme-object CONSTANT-0 #F)
                (Scheme-object CONSTANT-1 0)
                (equate new-bit-string-ref new-bitstring-ref-0)
                (word u #x303)
                (block-offset new-bitstring-ref-0)
                (LABEL new-bitstring-ref-0)
                (mov w (r 0) (@ro b 4 0)) ; eax <- bs
                (mov w (r 3) (@ro b 4 4)) ; ebx <- i
                (and w (r 0) (r 5))     ; remove tag
                (and w (r 3) (r 5))     ; remove tag
                (add w (r 4) (& 8))     ; pop bs and i
                (and w (@r 4) (r 5))    ; unmask return address
                (mov b (r 1) (& 31))    ; cl <- i & 31
```

```
            (and b (r 1) (r 3))
            (shr w (r 3) (& 5))     ; bs >> 5
            (mov w (r 0) (@roi b 0 8 3 4)) ; eax <- bs[i>>5]
            (shr w (r 0) (r 1))     ; shift selected bit to LSB
            (test w (r 0) (& 1))
            (jz b (@pcr is-false))
            (mov w (@ro b 6 8) (& #x20000000)) ; return #T
            (ret)
            (label is-false)
            (mov w (@ro b 6 8) (& 0)) ; return #F
            (ret)
            )
          new-bit-string-ref))
      '(define-macro (deflap name . lap)
         `(define ,name
            (scode-eval
             ',((access lap->code (->environment '(compiler top-level)))
                name
                lap)
              system-global-environment))))
  "inserting new-bit-string-ref code"))
```

5.4.5 Dispatchers and Couplers

We can now build our dispatchers from the predicates and actions above. We will build two dispatchers, since we have two different contexts of interest. The first dispatcher replaces `bit-string-ref` with `new-bit-string-ref`. Our coupler will pass it a combined context, produced by merging all 28 original contexts. Access to this merged context will allow the dispatcher's predicates to look for type consistency throughout the entire program. The second dispatcher inserts the lap code into the `propagate-nodes!` context. It therefore must be passed only that one context, and not the merged context.[‡‡‡‡‡] The predicate constructor `make-predicate/check-comb-param-types` is a default ViewForm predicate constructor that checks the combinations in the viewcode expression list. The combination's arguments must satisfy the passed-in predicates. In the predicate below, the first argument to the combination must be consistent with `*bit-string-type*`, and the second argument must be consistent with `*non-negative-int-type*`.

```
(define dispatcher/vf-new-bit-string
  (make-simple-dispatcher
   (make-predicate/check-comb-param-types
    (list (cons first  (make-user-predicate/type-consistent *bit-string-type*))
          (cons second (make-user-predicate/type-consistent *non-negative-int-type*))))
    action/vf-repl-bitstring-ref))

(define dispatcher/vf-new-lap-code
  (make-simple-dispatcher predicate/test-os action/vf-insert-lap))
```

[‡‡‡‡‡] While having two dispatchers is less convenient than having one, this is not problematic for this example. It does suggest, however, that the characteristic vform interface should take a list of contexts instead of a context.

We are now ready to write our coupler. As is typically the case, the control structure for the coupler must create the contexts, invoke the proper views on the contexts, compute the viewcode expression list, invoke the passed-in dispatchers on the appropriate viewcode expression lists, and register the coupling, if any. These tasks are implemented in the control structure constructor below.

```
(define (control/vf prim target-file rest-files)
  (lambda (vforms)
    (let ((new-bitstring (car  vforms))
          (new-lap       (cadr vforms)))
      (lambda (vform context vc-exps)
        (let ((contexts       (map make-context-from-file rest-files))
              (matrix-context (make-context-from-file     target-file)))
          (for-each (lambda (context)
                      (ensure-liar-view-computed!  context)
                      (ensure-alpha-view-computed! context))
                    (cons matrix-context contexts))
          (let ((merged-context (apply merge-contexts!
                                       (cons (merge-contexts matrix-context (second contexts))
                                             (cddr contexts)))))
            (ensure-platform-view-computed! matrix-context)
            (let ((prim-combs (list-transform-positive
                                  (append-map (lambda (exp) (exp->recvs exp merged-context))
                                              (var-binding->var-refs prim matrix-context))
                                (lambda (exp) (type/combination? (up-link exp))))))
              (if (and (do-vform new-lap       matrix-context '()))
                       (do-vform new-bitstring merged-context (map up-link prim-combs))
                   (register-coupling! merged-context vform))
              matrix-context)))))))
```

`control/vf` has been parameterized on a primitive, a file, and a list of files, respectively. The primitive (e.g., `bit-string-ref`) is the one whose references we are interested in replacing. The passed-in file is the one into whose context the lap code will be inserted and also the one searched for instances of `bit-string-ref`. The list of files contains all the other files to be included in the combination argument search on the instances of `bit-string-ref` that are found.

The control structure begins by creating contexts for each of the 28 files, then invoking the liar and alpha views on them. Afterwards, it merges the views into one combined view. The order is important: the liar view trades space for time, and the space requirements of invoking the liar view on one huge context are disproportionately large. Merging contexts, on the other hand, trades off time for space. This complementary tradeoff is a nice way to provide faster liar-view computation for smaller contexts, while still allowing the creation of larger contexts (via merging) with the available computational resources.

The control structure next computes the platform view on the matrix context (the one containing `propagate-nodes!`). We could have computed this view on every context, but that is unnecessary. The control structure then begins computing the viewcode expression list for the bit

string dispatcher. This list is constructed by computing a list of viewcode expressions receiving `bit-string-ref`'s value, in the matrix context since that is where we are interested in modifying the references to `bit-string-ref`. This list is then filtered for those viewcode expressions in the operator position of a combination. The bitstring dispatcher is invoked on this list and the merged context. The new lap dispatcher is subsequently invoked on the matrix context. If a coupling is rendered, it is registered with the view invalidation/recoupler. Finally, the matrix context is returned so that it can be unparsed or otherwise processed by the user or the compiler.

The coupler can now be implemented. It uses the control structure above, and the dispatchers created earlier. The variables `matrix-file` and `*vf-files*` contain the `propagate-nodes!` filename and the other 27 ViewForm filenames, respectively.

```
(define coupler/vf
  (combine-vforms (control/vf (lookup-predefined 'bit-string-ref) matrix-file *vf-files*)
                  (verbose-expl-combiner "integrating definitions")
                  (list dispatcher/vf-new-bit-string dispatcher/vf-new-lap-code)))
```

5.4.6 Platform View

We now turn to the implementation of the platform view. The platform view collects a description of the platform processor ViewForm is currently running on. It can also be set by the user, when the computing platform is to be something other than the underlying computing platform. This view is implemented the same way as the `*lambda-view*` and the comb view, via the following five steps:

1. Implement the merging and copying procedures, as well as the view initialization value
2. Implement the view dispatchers
3. Combine the dispatchers into a view
4. Register the view
5. Provide a way to invoke the view

For the first step, we must decide upon a representation for the view information. For the platform view, a string will suffice. The corresponding merge, destructive merge, copying, and initial value are therefore simple string operations. Since the semantics of combining two different platform views is otherwise dependent on the use of the view information, we make platform view merging default to the null platform.

```
(define (context/platform context) (retrieve-view *platform-view* context))
(define (set-context/platform! context info) (set-view! *platform-view* context info))
(define (platform-merge platform-view1 platform-view2)
  (if (string=? platform-view1 platform-view2)
      platform-view1
      platform-id))
(define platform-id "")
```

The next two steps are combined into one piece of code. For the platform view, we need only one action (and not a full dispatcher) since we are not doing a code walk or looking at any code. This action acquires the view information from `microcode-id/operating-system-variant`, an MIT Scheme predefined variable. This variable references a string containing information from which the processor platform can be gleaned.[§§§§§]

The third step is to create the view. This view is named `*platform-view*`, and is constructed using `*id-control-structure*`, a control structure that unconditionally invokes each passed-in vform on the passed-in context and viewcode expression list. This code is given below.

```
(define *platform-view*
  (make-view *id-control-structure*
             (verbose-expl-combiner "platform architecture information")
             (list (make-explained-action
                    (lambda (vform context vc-exps)
                      (set-context/platform! context microcode-id/operating-system-variant))
                    "storing the operating system variant"))))
```

The fourth step, registering the view, is straightforward, since we have previously defined all the necessary components:

```
(register-view! *platform-view* platform-merge platform-merge identity-procedure platform-id)
```

Finally, we provide a way to invoke the view, and define `x86-platform?`:

```
(define (ensure-platform-view-computed! context)
  (if (string-null? (context/platform context))
      (do-vform *platform-view* context '())))

(define (x86-platform? context)
  (there-exists? (list "x86" "386" "486" "pentium")
    (lambda (machine-string)
      (substring? machine-string (context/platform context)))))
```

5.4.7 ViewForm Output

The ViewForm coupler was invoked by evaluating: `(do-vform coupler/vf #f '())`. The resulting context is given below. The reference to `bit-string-ref` in `propagate-nodes!` was replaced by `new-bit-string ref`, and the lap code was added.

[§§§§§] If this variable was not available, a C function could be written and called to provide the information.

```
(define propagate-nodes!
  (lambda (tc-matrix)
    (let* ((size (vector-length tc-matrix))
           (tc-copy (make-initialized-vector
                     size
                     (lambda (idx)
                       (let ((result (bit-string-allocate size)))
                         (begin (bit-string-move!
                                 result
                                 (vector-ref tc-matrix idx))
                                result))))))
      (let loop-k ((k 0)
                   (up-to-date tc-matrix) (work-copy tc-copy))
        (let loop-rows ((row 0))
          (cond ((fix:= k size)
                 (if (not (eq? up-to-date tc-matrix))
                     (let loop ((index 0))
                       (cond ((fix:= index size))
                             (else
                              (vector-set! tc-matrix index (vector-ref up-to-date index))
                              (loop (fix:1+ index)))))))
                ((fix:= row size) (loop-k (fix:1+ k) work-copy up-to-date))
                (else (let ((row-bit-string (vector-ref up-to-date row)))
                        (begin (if (new-bit-string-ref row-bit-string k)
                                   (bit-string-or! row-bit-string
                                                   (vector-ref (or (and (fix:< k row)
                                                                        work-copy)
                                                                   up-to-date)
                                                               k)))
                               (vector-set! up-to-date row (vector-ref work-copy row))
                               (vector-set! work-copy row row-bit-string)
                               (loop-rows (fix:1+ row))))))))))))

(define-macro
  (deflap name . lap)
  (quasiquote
   (define (unquote name)
     (scode-eval (quote (unquote ((access lap->code
                                          (->environment (quote (compiler top-level))))
                                  name lap)))
                 system-global-environment))))

(define new-bit-string-ref
  (let ()
    (deflap new-bit-string-ref
            (entry-point new-bit-string-ref)
            (Scheme-object constant-0 ())
            (Scheme-object constant-1 0)
            (equate new-bit-string-ref new-bitstring-ref-0)
            (word u 771)
            (block-offset new-bitstring-ref-0)
            (label new-bitstring-ref-0)
            (mov w (r 0) (@ro b 4 0))
            (mov w (r 3) (@ro b 4 4))
            (and w (r 0) (r 5))
            (and w (r 3) (r 5))
            (add w (r 4) (& 8))
```

```
        (and w (@r 4) (r 5))
        (mov b (r 1) (& 31))
        (and b (r 1) (r 3))
        (shr w (r 3) (& 5))
        (mov w (r 0) (@roi b 0 8 3 4))
        (shr w (r 0) (r 1))
        (test w (r 0) (& 1))
        (jz b (@pcr is-false))
        (mov w (@ro b 6 8) (& 536870912))
        (ret)
        (label is-false)
        (mov w (@ro b 6 8) (& 0))
        (ret))
    new-bit-string-ref))
```

To demonstrate the user interaction, the line `(propagate-nodes! (vector (vector 1 2 3)))` was added to the code. The view invalidation/recoupler was then invoked. This produced the following instance of user interaction. When the user responded with `y` (not necessarily correct), ViewForm produced the same context expressions as above.

```
The expression: [16] (vector 1 2 3)
     returning types: (#[uninterned-symbol 17 vector-type19])
     is inconsistent with the required type: #[uninterned-symbol 18 bitstring-type30]
     for the expression to be transformed: row-bit-string
Shall I accept this and continue?
Enter y to accept or n to reject: y
;Value 19: ((#[%context 20]))
```

5.4.8 Quantitative Measurements

The following tables contain the same measurements collected for the previous examples. These measurements are discussed next.

Pedigree Example	Number of Constructs	Total Number of Lines	Average Number of Lines per Construct
Predicates	3	21	7
Actions	2	55	27.5
Dispatchers	2	8	4
Rules	1	27	27
New Views	1	22	22
Other	N/A	0	N/A
Total	9	133	N/A

Number of Lines the Action Modified	38
Number of Other Lines the Action Depended Upon	0
Number of Lines Analyzed	7616
Time to Couple Code	92.0 seconds
Time to Recouple Code	100.2 seconds
Time to Compile Code	230.4 seconds

5.5 Analysis and Evaluation

For each of the three examples, the two evaluation metrics given earlier are assessed. The first metric is whether ViewForm successfully generated the desired coupled code (or its equivalent) and the second is whether the desiderata from Chapter 2 were met. Afterwards, I discuss what I learned from my experience with view-based abstraction as well as various important issues that came up during my experimentation.

5.5.1 The Desired Couplings Metric

The desired couplings metric has two possible values: true or false. It is true if the ViewForm-generated code is either identical or equivalent to the code desired by the programmer. Equivalence is determined with respect to the criteria the programmer set out when developing the desired code. For instance, in the dispatch example, the criterion is to transform a linear-time process to an expected constant-time process. According to this criterion, the code in Figure 1-4 is equivalent to the code in Figure 1-8, even though they are not syntactically the same. This criterion could also have been satisfied by code using a different expected constant-time dictionary data structure.

5.5.1.1 The Simulator Example

For the simulator example, the desired criterion is to generate a particle data structure implementation that uses flovecs to represent particle values. The desired code (which meets this criterion) was given in Figure 5-2 and the ViewForm generated code was given in Figure 5-3. These two pieces of code are not syntactically equivalent. Both, however, do have identical interfaces. While this is enough to satisfy the criteria, let us explore the two differences between these pieces of code for completeness. The first difference is that the desired code uses the syntactic sugar:

```
(define (<proc> <formal-1> … <formal-n>)
  <body>)
```

while the generated code uses the form:

```
(define <proc>
  (lambda (<formal-1> … <formal-n>)
    <body>))
```

This is a purely syntactic difference: the two forms are completely semantically identical. Thus, this has no negative bearing on the desired coupling metric. The second difference involves ß-reduction (see Figure 3-3). A Scheme expression of the form:

```
((lambda (particle) (vector-ref particle 3))
 particle)
```

can be ß-reduced to the form:

```
(vector-ref particle 3)
```

The two forms above are semantically equivalent. In fact, compilers are free to perform ß-reduction (as per Figure 3-3) as a way of statically optimizing out certain procedure calls. Since the non ß-reduced expressions in the generated code can be ß-reduced, this difference has no negative bearing on the desired coupling metric. The desired and generated code are therefore equivalent. This validates the desired coupling metric.

5.5.1.2 The Pedigree Example

For the pedigree example, there are two desired criteria. The first is substituting `*` with `sc-*` in `P_pedigree`. The second was producing a version of `P_pedigree` in which `p-sum` and `p` were inlined. For the first criterion, the desired code in Figure 5-9 is compared to the generated code in Section 5.3.7. The generated code differs from the desired code in two aspects. The first aspect is the `define` expression syntactic sugar described in the preceding section, in the simulator example. The second aspect is in the difference between:

```
`(,<exp-1> … ,<exp-n>)
```

and

```
(quasiquote ((unquote <exp-1>) … (unquote <exp-n>)))
```

Like the `define` expression syntactic sugar, these forms are completely equivalent. One is just syntactic sugar for the other. Since there are no other differences between the desired and generated `P_pedigree` code, the desired coupling metric is validated for `P_pedigree`.

Next, `p-sum` and `p` must be examined. The criterion is to inline `p-sum` and `p` into `P_pedigree`. This is exactly what `define-integrable` stipulates to the MIT Scheme syntaxer. `p-sum` and `p` were, however, changed somewhat. Specifically, an extra non-ß-reduced combination was added to `p-sum` and `p`. The purpose of this addition was solely to force the MIT Scheme syntaxer to substi-

tute the arguments for the formals exactly once. The additional combination does not adversely affect the inlining of `p-sum` and `p`, especially since a compiler could easily decide to eliminate it post-inlining. Thus, the `p-sum` and `p` code is equivalent to the desired code. This completes the validation of the desired coupling metric for the pedigree example.

5.5.1.3 The ViewForm Example

For the ViewForm example, one objective was to replace a reference in `propagate-nodes!` to `bit-string-ref` with `new-bit-string-ref`. This objective was achieved, the only other difference being the syntactic sugar from:

`(define (propagate-nodes! …) …)` to

`(define propagate-nodes! (lambda (…) …)`

As discussed earlier, this is a purely syntactic difference. The second objective was to add the lap code. This code was also added successfully, the only difference being the backquote symbol (i.e., `), unquote symbols (i.e., ,) and quotation symbol (i.e., ') were replaced by `quasiquote`, `unquote`, and `quote`, respectively. As previously discussed, this equivalence is also purely syntactic. Since the differences between the desired and generated code are all purely syntactic, we can validate the desired metric coupling for the ViewForm example.

5.5.2 The Desiderata Metric

The desiderata metric consists of five properties that view-based abstraction must possess. Each property is discussed below, with respect to the simulator, pedigree, and ViewForm examples.

5.5.2.1 Backwards Compatibility

None of the examples required modifications to the source language, modifications to MIT Scheme's implementation or modifications to the source programs (i.e., pragmas or annotations). None of the examples required rewriting the source programs into a different language. In all of the cases, the need for direct source code access was limited to the code being modified or the code that the modification depended upon. This means view-based abstraction was fully backwards compatible in the examples with respect to the criteria defined in Section 2.3.1.

5.5.2.2 Incrementality

As discussed earlier, incrementality is assessed by comparing the scope of the desired modification with the scope of the coupling. For the simulator example, the desired modification changed three `define` expressions: `gunk:initial-value`, `gunk:particle-value`, and `gunk:set-particle-value!`. The changes were dependent upon the one line implementing the vector access

in gunk:particle-value. This scope is the same as that of the corresponding actions, which modify the three define expressions and depend on the implementation of gunk:particle-value. In addition, as shown in the table from Section 5.2.8 , the number of lines modified and depended upon by the action is low compared to the size of the simulator code.

The desired pedigree example modification depends upon the definitions of P_pedigree, p-sum, and p. The actions we defined for the pedigree likewise depend on the definitions of these three procedures. Our sc-* predicate, however, requires a data-flow analysis on P_pedigree. While this may seem to indicate that our predicate has a larger scope than what is required, this analysis need only look beyond P_pedigree to look for combinations whose operator is *'s value. This additional scope is optional. predicate/sc-* could easily be modified to look for operators that *are* textually identical to *. This does not require any sort of data-flow analysis, but relies on the presence of the usual-integrations declaration. The additional scope was optionally implemented to make the predicate less fragile.

The ViewForm example modification depends on the code in propagate-nodes!, as well as any code that calls propagate-nodes!. The actions we implemented likewise depend on the same code, although the scope was extended to include files that would not otherwise be allowed to call propagate-nodes!. This extension of scope was done solely to demonstrate ViewForm's performance on a larger program, and not out of necessity.

These three examples demonstrate the property of incrementality. Our couplers did not require a scope larger than that of the desired modification (the extra scope in the pedigree and ViewForm cases was not required). Furthermore, our couplers did not depend on any other potential couplings in the code, whether or not they were ViewForm mediated.

5.5.2.3 Language Independence

Since all three examples were performed using ViewForm (which takes only MIT Scheme as input), language independence is not easy to demonstrate. In fact, some couplers depended upon three specific MIT Scheme features (i.e., (declare (usual-integrations)), define-integrable, and microcode-id/operating-system-variant). These dependencies are not necessary, they are simply convenient. While view-based abstraction does not depend on any language feature or programming paradigm, a more empirical test would be to implement a variety of ViewForm versions that do not take Scheme as input. Instead, I will argue view-based abstraction's language independence based on the fact that the view-based abstraction specification in Chapter 3 does not require any specific language features.

5.5.2.4 Ease of Understanding and Usability

Ease of understanding and usability cannot be measured objectively, although the measurements taken on the simulator example in Section 5.2.8 are a good starting point. These measurements show that modifying 14 lines of uncoupled code took 65 lines of ViewForm code. This results in a ratio of 4.6 lines of ViewForm code per line of modified code. While a lower ratio would certainly be better, the real gain is that future programmers will not be required to pay for the consequences of broken modularity. When modifying code with implementation couplings under black-box abstraction, future programmers must search for and find existing implementation couplings, must determine what their preconditions were, must determine whether those preconditions are still valid (i.e., whether the couplings affect the future programmer's desired modifications), must determine how to revert or patch those couplings (and thereby recursively do this all over again for each reversion or patch), and must test the reversions or patches. This price for broken modularity is high, and furthermore must be paid each time a future code modification is made to any affected module. The programmer who is implementing the simulator example implementation coupling can eliminate these future costs with 65 lines of ViewForm code - code that this programmer can write more easily since he/she is the one who developed the implementation coupling to begin with. Compared to the recurring costs of broken modularity, 65 lines of ViewForm code is inexpensive.

Let us proceed to examine the 65 lines of ViewForm code for the simulator example. Over a third of the ViewForm code is in the actions. The action code is usually the easiest to express, since it consists mostly of some sort of code template. Another third of the ViewForm code is devoted to the coupler. This code is also straightforward to write, as can be evidenced by the fact that the same approach and structure was used to write every coupler presented in this dissertation (i.e., create a context, compute views on the context, build up the viewcode expression list, invoke the rules/dispatchers on the viewcode list, register the coupling). The most difficult code to write, the predicates, make up roughly 12% of the simulator example ViewForm code. The simulator example predicates, however, are only two lines long. This can be attributed to the use of complexity layering via the ViewForm code library.

The pedigree example shows similar kinds of numbers. The ratio of ViewForm lines of code to modified lines of code is 4.4, a slightly lower number than in the simulator example. Some of this difference can be attributed to the larger size of `P_pedigree`. The comb view code comprises 23% of the total ViewForm code. This code is not specific to the pedigree example, and is thus reusable. The actions make up 18% of the ViewForm code. Interestingly enough, the number of action code lines is smaller than the number of modified lines. This is because several instances of `*` in `P_pedigree` were substituted by the same `sc-*` action. The rules and the

coupler comprise 36% of the code, mainly because there are two rules and one coupler. The predicates comprised 18% of the ViewForm code. At least two of the predicates (`predicate/no-mutations` and `predicate/sc-*`) can be made general enough to be in the ViewForm library. `predicate/no-mutations` already is general enough, and in `predicate/sc-*`, `*` can be made a parameter. This change would have reduced the predicate line count by half. Adding these predicates to the ViewForm library is beneficial because predicates are generally more difficult to write than other view-based abstraction constructs. Any ViewForm library functions that can be used to simplify the predicate-writing process will disproportionately simplify the implementation of the corresponding coupling.

For the ViewForm example, the ratio of ViewForm lines of code to modified code is 3.5. This is because more lines of code are modified in this example, thereby amortizing the total number of ViewForm code lines. This also explains why over 40% of the ViewForm code lines were in the actions. The remaining lines of ViewForm code are similar to what was measured in earlier examples, and the average number of lines per predicate is low as desired.

The measurements taken on the ViewForm code, together with the approaches and techniques used to write the code, share some common aspects. These shared aspects include ways of combining predicates or actions, ways of passing viewcode expression lists, and ways of implementing control structures and views. These commonalities can form the foundation for programming skills that more quickly lead to finished couplers. I found this to be consistent with my experiences writing couplers, and I expect other programmers to have similar experiences.

5.5.2.5 Amortizable Time Savings

The times needed to generate a coupling and the number of lines analyzed for each example are given in Sections 5.2.8, 5.3.9, and 5.4.8. The simulator example produced a rate of approximately 126 lines per second. By comparison, the MIT Scheme compiler produced a rate of 43 lines per second, or about a third the rate of ViewForm. For the pedigree example, ViewForm's rate was 131 lines per second and the MIT Scheme compiler's rate was 34 lines per second.

One difference in the ViewForm example is the time needed to couple the code. ViewForm achieved an average rate of approximately 83 lines per second, less than in the previous examples. The ratio of coupling time to compile time was also worse; about 40%. The explanation for this is the extra time needed to merge the 28 contexts. Thirty percent of the coupling time was devoted to this merging effort, which was otherwise insignificant in the previous examples. It is important to note, however, that these measurements are worst-case numbers and are artificially inflated. They are worst case in that the liar view was fully computed over every expression in

every context in the entire ViewForm program. They are artificially inflated in that only five contexts were actually needed to compute the coupling, since only those five could call `propagate-nodes!`. This would have led to a ViewForm time of 18 seconds, with a rate of 105 lines per second, and a compilation time of 79.5 seconds, and 24 lines per second. Nevertheless, one goal of this example was to demonstrate ViewForm's performance on worst-case scenarios. Towards that end, it is clear that increasing the size of the program an order of magnitude did not create a disproportional increase in the coupling time or decrease in the coupling rate (either absolutely or compared to the compilation rate). Coupling the code, therefore, would not add a substantial time burden to the compilation of any example.

Given these rates, it could be argued that a huge (i.e., million-line) program would not be suitable for ViewForm. This is true only if all million lines must be analyzed. If validating a precondition requires analyzing a million line program, however, the corresponding implementation coupling is unlikely to be applied by either a human being or a machine. Instead, experience dictates that modularity can severely reduce the amount of code that must be analyzed. It is more likely that only a few modules need to be analyzed, either because the scope of an implementation coupling is limited, or because a programmer decides that only a few modules are actually relevant to the coupling. This was the case with the simulator example, where out of an application spanning over 10,000 lines of code, roughly 11% were relevant and needed to be analyzed.

Another way ViewForm couplings can save time is via the automatic code analysis. In order to safely apply the desired simulator modifications, for example, the simulator must be analyzed to ensure that every particle value is a floating point number. Doing this analysis manually can take substantially more time than 92 seconds, and can be prone to human error. Even one error can create a bug that takes minutes or even hours to solve (e.g., if the bug rears itself later, because of an unrelated program modification). This leads to the next form of saving time.

By eliminating the possibility of certain bugs, ViewForm couplers can reduce debugging time. In the simulator example, our coupler will never modify the particle value representation if even one, hard to find use of `gunk:particle-set-value!` sets a particle value to something other than a float. Predicates can alert programmers to the existence of these kinds of conditions by displaying messages, for example.

These observations are consistent with the claim that view-based abstraction is easy to understand and use, and can save time. Of course, as is the case with most new models, acquiring practical experience with view-based abstraction goes a long way towards enhancing understanding, usability, and the realizable time savings.

5.6 Lessons Learned

In this section, I discuss some important aspects of my experiences using view-based abstraction and ViewForm.

5.6.1 Determining Preconditions and Program Modifications

The difficulty of determining a set of preconditions is related to the desired level of fragility. Strong preconditions are easy to develop and implement, while weak preconditions are difficult to develop because they must take future changes to a program into account. For example, under black-box abstraction, a programmer may decide to increase a loop's performance with a macro that copies a function's implementation directly into a tight loop. At first glance, this macro may seem like an easy solution to the implementation coupling problem: when the selector's code changes, so does what gets copied into the loop. No apparent preconditions seem necessary. Unfortunately, the situation is more complex. The code cannot be copied if it is later changed to maintain local state or modify state through a variable that is not lexically visible inside the tight loop. Even worse, any free variables in the selector code may be "captured" by variables in the tight loop's lexical scope.[******] ViewForm does not solve the hard problem of automatically determining robust preconditions. Rather, ViewForm's job is to make the preconditions (and modifications) easy to express. In any case, care and caution are needed when developing less fragile or weaker preconditions.

Once expressed, ViewForm can significantly assist in validating the preconditions. These preconditions can involve whole-program analyses such as data-flow analysis, control-flow analysis, side-effect analysis, and alias analysis. Under black-box abstraction, programmers typically perform these analyses manually, even though a computer can automatically perform many of these analyses to varying degrees of precision. Manually computing these analyses and manually validating the preconditions can be complex and error prone. Even partial assistance (i.e., validation that depends on user interaction) can reduce complexity and potential sources of error.

5.6.2 Conservative Program Analyses

As mentioned earlier in Section 5.2.7, accurate data-flow analysis can be hampered when values flow in and out of data structures, closures, and top-level variable names. This problem can preclude the simple and straightforward expression of a weak precondition. The root of the problem lies in the consequences of the theory that not all program properties are computable

[******] While hygienic macros[15] can prevent this kind of variable capture in scheme, languages like C do not provide this kind of facility.

in general.[44] This means that ViewForm cannot automatically compute any program property with perfect precision in general. Data-flow analysis falls into this category.

I suggest three approaches when precondition expression is precluded this way. The first is user interaction. As suggested in Section 5.9.0.1, user interaction provides full backwards compatibility by allowing views to "compute" program properties based on user input. The second approach is to widen a predicate's scope. For example, predicates can be implemented to look at all the viewcode expressions where a value of interest goes into or comes out of instead of starting at one expression along the way and branching out from there. The third is inserting dynamic checks into the viewcode (e.g., dynamic type checks). While this problem can be mitigated in other ways, ultimately the ease by which a precondition can be expressed depends on the availability of an acceptable amount of precision in view information.

5.6.3 Default Program Analyses

I found it useful to provide, by default, an alpha view and a data-flow view. These alone allow for the expression of a variety of preconditions. These views can also be useful during program development. While developing ViewForm, I extensively benefited from the ability to find unbound variables and parameter lists inconsistent with call sites. Other useful views compute control flow, dynamic type, side effect, and alias information, although they were not included in ViewForm by default.

5.6.4 Default Coupling Library

Programmers tend to attempt certain optimizations more frequently than others. These include inlining, operator strength reduction, limited partial evaluation, code motion, representation shifts, and semantics-preserving code rewrites (i.e., using language constructs "known" to be more efficient). In fact, few programmers seem to be able to describe more than 15 or 20 kinds of optimizations they perform on a regular basis. One can conclude that a relatively small (of size 30-40) library of optimizations would be useful in most implementation-coupling circumstances. Building a library of these constructs is an important factor in the complexity layering argument presented in Chapter 4. I found that even with ViewForm's limited default vform library, I was able to reuse library components on a regular basis.

5.6.5 View Selection and Computation

Views are undoubtedly the most computationally expensive part of ViewForm. This observation is consistent with that of similar research efforts.[28] This is not surprising, as the more interesting views are based on interprocedural analyses. This means that view selection is very important: choosing a view with more precision than necessary can adversely affect perform-

ance. It is also important to reduce the size of the contexts over which views are computed, since non-local analyses often scale exponentially in the size of their input. I found that views did not generally need to be computed over an entire application. Instead, views could generally be invoked on pieces of code several hundred or thousand lines long even though the overall applications were 5 to 10 times larger. This reduced the amount of time spent computing views to or below acceptable levels. Contexts were instrumental in allowing these reductions to be made.

An important point is that if the amount of code that needs to be analyzed grows slowly with respect to the size of an application, then views will remain viable even for larger applications. For this to be true, module size, at least, must grow more slowly as the size of an application increases. This is likely to be true, as the purpose of modularizing an application is to break it into small pieces amenable to human understanding. Since the capacity of human understanding does not grow with the size of an application, it stands to reason that module size will not either.

5.7 View Updates

Ideally, views are updated at a fine grain; the finest grain being every time the program is modified. From a practical standpoint, this is not desirable for moderately sized programs, where gratuitous view computation may consume an unbearable amount of resources. However, not all program modifications leave a context in a state consistent with its current views. For this reason, ViewForm permits vforms to manually control when views are updated. This control is especially important when view computations and non semantics-preserving program modifications are interleaved. This is because, in general the viewcode needs to be in a consistent state for a view to be updated correctly and views must be in a consistent state for viewcode to be modified correctly. This control is also important for performance. For example, a set of rules that each use a context's liar view but do not invalidate it can be combined without requiring interleaved view recomputations.

5.7.1 View Consistency During Rule Invocations

One concern during a rule's execution is maintaining the consistency of the views used by its sub-vforms. The potential problem is that actions may modify viewcode expressions, invalidating a view's information, but not update the views. This alone is not sufficient to cause a problem. It must also be the case that a subsequent predicate depends on the validity of the now invalid view information.

In practice, there are various ways to address this problem in ways that are not too expensive. One is an incremental view update algorithm. This is the most useful approach, but may not be possible for every program analysis algorithm employed by views. Another approach is to minimize the frequency of view recomputation by combining rules into compound rules that do not need recomputation or need infrequent recomputation. This is similar to how compiler phases are designed. A third solution is to update view information conservatively. For example, the liar view could be incrementally updated with less precise, but nevertheless correct information. Full recomputation for more precision could be deferred until actually needed.

5.8 Source Code Maintenance

One possible source of complexity is the extent to which a programmer modifying the original uncoupled code need be aware of any previously applied couplings. The more this programmer must deal with existing couplings, the more complexity this programmer must manage while making new program modifications.

To analyze this situation, the space of couplings can be divided into two categories: *interface preserving*, and *interface non-preserving*. An interface non-preserving modification is one that alters an interface's specification. An interface-preserving one does not. Note that semantics preserving does not imply interface preserving (although the converse does hold). In real-time systems, for example, altering the performance of an implementation with a semantics-preserving modification can violate an interface's hard time constraints.

5.8.1 Interface Non-Preserving

Suppose we are in an implementation-coupling situation as in Figure 1-6. Suppose a coupling implements a required part of the coupled module's interface. This means the coupling is an interface non-preserving coupling, since it makes the interface valid. A programmer modifying the uncoupled source may inadvertently invalidate the coupling's applicability, resulting in a non-functional module once the view invalidation/recoupler runs. Maintainability in this situation can be enhanced by ensuring that the interface non-preserving coupling's predicate alerts the programmer when it is no longer applicable. A user-friendly predicate could go so far as to explain why it is no longer applicable. This approach allows a programmer to make changes without considering the effects on such a coupling. The programmer must worry only about the coupling when he or she is alerted to a problem by the coupler's predicate.

5.8.2 Interface Preserving

A coupling's effects cannot be determined as easily in this situation as in the non-preserving situation. In the non-preserving situation, the coupling's effect was to help implement the interface; no coupling, no valid interface. The purpose of an interface preserving coupling, on the other hand, is usually to increase performance. These kinds of couplings will not necessarily have the intended effect when the original uncoupled code is modified in some other way.[††††††] Precisely and accurately capturing every detail under which a coupling will favorably alter performance is also likely to be very difficult and not necessarily fruitful. This is a general problem not specific to view-based abstraction. Due to the complexity of predicting the performance characteristics of any program modification, programmers typically resort to profiling or simple timing runs to empirically determine the effects of various changes.

Even in this case, couplings that cannot be reapplied due to subsequent code modifications can still alert the programmer to that fact. Performance may or may not be affected, and may or may not increase or decrease. This is still a markedly better situation than what happens under black-box abstraction, where a programmer can invalidate an implementation coupling without realizing it. The most important point about performance-altering couplings is that, unlike the non-preserving case, a programmer need not worry about "fixing" a coupling that did not succeed. A coupling need be considered only if performance drops to unacceptable levels; the application will still function since the coupled module's interface remains valid.

5.9 User Interaction

Users can interact with view-based abstraction constructs in two important ways: during the coupling process (step v) and during view computation (step iii). The former most likely happens when a predicate cannot completely validate a precondition. A user can be asked whether to proceed anyway, or can be asked for additional information. Views can also ask the user for help. This may happen, for example, when a view runs out of a resource that a user can replenish.

User interaction, unfortunately, can break safety. Users are not infallible and can introduce incorrect assumptions or cause incorrect program modifications. These problems are, however, pervasive throughout every programming exercise in which a person is involved. The more specific issue is whether user interaction can disproportionately create problems under view-based abstraction.

[††††††] Performance effects cannot be accurately predicted in general.

To address this issue, let us examine two possibilities. These two cases are also possible without view-based abstraction, when a user is manually computing the preconditions:

- An action that should not run gets run (false positive predicate)
- An action that should run does not get run (false negative predicate)

An interface-preserving action will not cause errors for false negatives, but may for false positives. An example of the former action is one that rewrites code to use special forms better compiled by the compiler. An example of the latter is the simulator actions. Thus, extra effort is well spent ensuring that the latter kind of actions have adequate predicates dispatching them.

An interface non-preserving action can cause errors for either false positives or false negatives. This is the main reason I do not advocate mixing interface non-preserving actions with user interaction.

User assumptions can also be problematic under view-based abstraction in that they must be rechecked when the view invalidation/recoupler is run. This implies that each user assumption must be re-verified every time an invalidation occurs. There are one general and two specific ways to mitigate this problem, depending on the specifics of the circumstances. The general way (which I have not experimented with) is to develop a *user view* that caches the user's answers. When a predicate requests information from the user view, the cached information is returned if available. Otherwise, the user view asks the user directly. In this way, a predicate can be insulated from the user interaction, making the predicate easier to write.

Within this user-view approach, two specific circumstances exist. The more favorable circumstance is when the user's assumptions are easily checkable but not necessarily easily computable (or not computable at all). In this case, user interaction need only be requested during recoupling if the user assumption check turns up negative. This user assumption check can also be pushed to run time, by having a coupling insert checks (e.g., type checks, range checks, consistency checks) into the coupled code that validates the assumptions. This kind of user interaction can be quite beneficial, while at the same time fully retaining safety.

The more difficult case is when the user's assumptions cannot be easily checked. In this case, interface non-preserving couplings should not invoke a user view that caches responses, as this could lead to errors. While an automatic solution is more favorable under these circumstances, view-based abstraction is still a marked improvement over black-box abstraction, where user assumptions are implicit and usually undetectable in the code. Thus, in the worst case, when a user assumption is a necessary precondition for a program modification, view-based abstraction allows the assumption's existence to be expressed and available to the future programmers maintaining the code.

5.9.0.1 User Interaction Reduces Fragility

Given the preceding discussion of user interaction, it is fair to question whether user interaction really provides enough benefits to offset its potential costs. The answer to this question is that user interaction can significantly decrease a predicate's fragility. Since not all preconditions are easily (or possibly) computable, user interaction permits these preconditions to be expressed by predicates that rely on user-supplied assumptions. This, in turn, increases the space of expressible validating conditions beyond what is computable. Since preconditions are not necessarily computable, user interaction can be utilized to express any human-computable precondition.

5.10 Source Code Availability

Thus far, all view-based abstraction scenarios have implied direct, full source-code availability. While view-based abstraction does not require this level of access, less access can reduce the scope of expressible couplings. Let us consider a few scenarios in which the source code is not fully available.

5.10.1 Source Code Somewhat Available but Partially Hidden

The source code can be partially hidden for a variety of reasons. The source code may include trade secrets, it may be too large for local storage, it may be under RCS control, etc. Let us simply assume that the source code is available to some degree, but complete access is not possible.

To handle this case, ViewForm could be augmented by explicitly separating view computation from coupling application and view invalidation/recoupling invocation. In essence, this creates a client-server model where the server is a "view server" and the client is the computational substrate that manages everything else. The server can selectively block access to any part of the source code it sees fit, but may allow properties of the source code to be exported as views.

Within this augmented implementation and model, there exist two major, adversely affected aspects of view-based abstraction. The first is the quality and amount of view information. The view server may not have full access to the source code to begin with; this can lead to less precise or vacuous view information. The view server may also not be able to invoke arbitrary, user-defined views. Thus, predicates using locally implemented views that spend extra computational resources computing quality information may fail. Predicates that examine or navigate through the source code representation may also fail to return what would otherwise be a true value. Nevertheless, view-based abstraction can still be used to some degree; it just degrades depending on the particular couplers being used.

The second adversely affected aspect is the number of coupling modifications that are expressible. Any modification that requires source code, such as inlining,[‡‡‡‡‡‡] may not work under these circumstances.

5.10.2 Not Available and Not Hidden

For this situation, let us assume that the source code is not available, but the object code or some lower-level representation is available (after all, the computer needs something to run). While this seriously hampers the utility of view-based abstraction (black-box abstraction implementation coupling is also hampered in this case), view-based abstraction can still be useful depending on the kinds of view information that can be computed from the object code. If the object code's interface is known and trusted, then views can be based on the interface. Views can also be based on implementation details that have been (experimentally) deduced from the interface.

5.11 Fundamental Insights

My experience with view-based abstraction shows that an approach based on program transformations is well matched to the way programmers tend to perform couplings (i.e., using a text editor to transform a program). The six-step implementation-coupling process is based on the notions of preconditions and program modifications, both of which must be expressed in a transformation language. Preconditions depend on views, and views can be asked to answer any number of questions. I believe that the success of view-based abstraction firmly depends upon the extent to which views can answer these questions. This ties back to the issues of backwards compatibility and incrementality.

The reason I required backwards compatibility and incrementality in view-based abstraction is a direct consequence of view-based abstraction's heavy dependence on views. It would be much more straightforward to develop a view-based abstraction implementation under a contrived language where many interesting programming questions could be more easily answered. We could tailor Scheme this way. First, we eliminate first class procedures, continuations, and side effects. This eliminates imprecisions caused by calls sites with multiple operators, arbitrary control paths, and formal parameters with multiple values per procedure call. Then, we make Scheme checkably strongly typed, with checkable module and data abstraction specifications. This provides more information about types and interfaces. These changes allow views to be

---

[‡‡‡‡‡‡] A programmer can still, however, use specially customized code in place of the hidden inlined code so long as the programmer is certain the special code validly implements the hidden code's interface. Random number generator functions and transcendental functions are two obvious examples where this can happen beneficially.

more precise more cheaply. This approach, unfortunately, would have skirted the central issue of how programmers use interesting, often difficult to compute, program properties to couple implementations.

To address this central issue, I first note that determining many interesting program properties requires non-local program analyses. Many transformation systems do not include these kinds of analyses for three reasons. One is that these analyses generally consume large amounts of computational resources. Another is that the analyses are not always precise. The third is that they are not easily semantically characterized. Yet programmers manually perform these analyses many times during program development, debugging, porting, and optimization. This is precisely why I included support for non-local analyses as part of ViewForm.

To continue with the central issue, we must accept that not all interesting program properties are computable, or even computable given a reasonable amount of resources. While this is easy to accept, the consequences may not be. For an implementation coupling program transformation system to be useful, it must therefore depend on user interaction to help when views are asked to compute the otherwise non-computable. Since users can be wrong, any semantic attempts to characterize such a transformation system are now in jeopardy. Nevertheless, user interaction does provide two extremely useful properties. The first is the ability to narrow the scope of a program analysis. The second is incrementality, or the ability to use view-based abstraction on some parts of the program and not others. User interaction, thus, is essential in reducing a view's computational resource requirements and in allowing users to bypass the non-computability issue in any situation.

I believe the future of view-based abstraction, therefore, can strongly benefit from research in non-local program analysis methods. Algorithms that can answer more questions, with a greater variety of precision and computational resource tradeoffs, and with less user interaction, will make it easier to express preconditions and will further increase the robustness of the automatic coupling process.

Chapter 6

Related and Future Work

There exists a large body of research related to implementation coupling, views, and program transformation systems. The more closely related work is discussed below in this section. Future directions for view-based abstraction research are discussed afterwards.

6.1 Views

In general, *views* refer to different ways of presenting the same information. For example, a point in two dimensional space can be represented in Cartesian or polar coordinates. Each of these representations corresponds to a different view on the point. This section describes systems that use or provide views. For each case, the views are related back to ViewForm views.

6.1.1 The View Oriented Model

The View Oriented Model (VOM) [36] is the design of an environment of cooperating tools that can share data via a collection of views. Views in this system are different interfaces to data shared between tools. One of the main thrusts of this work is to show how views can be used to share data among independently written tools. Consistency is maintained by procedures associated with specific views and data. These procedures are activated when the data is modified. The system is implemented in a strongly-typed, object-oriented style.

VOM treats the data in its environment as a database made up of data-representation descriptions that are provided by the tool implementors. The descriptions are specified as combinations of a base set of primitive data types provided by the VOM. The descriptions are used to statically and automatically generate an implementation of the database and its concrete data representations. New views can be added to the database if necessary.

VOM is restricted by a limitation in the data types that can be combined to form objects with views although in later work [40], this limitation is overcome. On the other hand, VOM's model of maintaining consistency between arbitrary views in a transformation system is not prac-

tical in ViewForm for at least two reasons. The first reason is that local changes in one view can engender global changes in others, and computing these changes can be computationally intensive. The second reason is that transformations may not need fine-grained consistency between all active views, in which case maintaining consistency leads to wasteful computation. In addition, maintaining consistent multiple views can lead to incomprehensible code. This can happen, for example, if pervasive and complex changes are made to the data-flow view and are then automatically propagated back to the text view. For these reasons, VOM views would not be practical in ViewForm.

6.1.2 PECAN

PECAN [72] is a programming environment that renders multiple views of a program for a user. The objective of PECAN is to show that modern computational power and graphics technology are capable of providing multiple views of a program in development that are updated consistently in real time. PECAN supports three kinds of views; *syntactic*, *semantic*, and *execution*. Unlike ViewForm views, the PECAN concrete views are rendered graphically to the user of the PECAN programming environment.

PECAN's syntactic view is similar to ViewForm's viewcode view. The syntactic view is rendered through an editor that supports a mix of a traditional character-based view and a structure-oriented view. Four of the supported semantic views include the symbol table view, the data-type view, the expression view, and the flow view. These views also provide information not unlike what is provided by ViewForm's viewcode, alpha, and liar views. The symbol table view is a hierarchical representation of the environment structure from the point of view of a particular expression. The data-type view is a representation of a data type being edited by the user. The expression view is a representation of the abstract data tree for an expression. The flow view displays control-flow information on the program in a form based on Nassi-Shneiderman flow charts. Other views that were not implemented are a dataflow view, a view that focuses on module-level abstraction, and a declaration view.

The third kind of view in PECAN is the execution view. Execution views are representations of the execution of a program. The two execution views supported by PECAN are the feedback and stack views. The former shows the statement being executed and the latter shows the execution stack as a series of stack frames. ViewForm does not provide these kinds of views, although they could be added.

Unlike ViewForm views, when users make changes in some of the PECAN views, PECAN automatically propagates the changes to other appropriate views. The default views implemented in PECAN can all be updated incrementally, thereby making the programming environ-

ment more responsive. In addition, PECAN is flexible enough to allow new views to be added in a modular way. PECAN's views are designed to be graphically rendered and efficiently updatable. PECAN also depends on the existence of a method for computing the incremental changes on all appropriate views given a small change to each specific view. These dependencies and requirements are necessary for new views, even though they are non trivial to comply with for arbitrary non-local program information. In addition, compliance is not always useful or necessary for specific implementation couplings. Using PECAN's view design for ViewForm would therefore limit the kinds of new views that could be added to those that can be incrementally updated, and would require extra code to implement the updating procedures. This is why the PECAN view design cannot be used to implement ViewForm views.

6.1.3 Semantic Program Graphs

A Semantic Program Graph (SPG) [64] is a canonical program representation that allows various views of a program to be easily derived and incrementally modified when a change is made in one of the views. The purpose of the SPG design is to show how a general program representation can also be used to accommodate views. SPG's are designed to allow the representation of different language paradigms (imperative, functional, and dataflow), parallel constructs, non-deterministic computation, strict and non-strict parameter passing, control-flow and dataflow information, and naming scopes. In an SPG, nodes represent operations and edges represent paths along which tokens can flow. Edges can have fan in or fan out, and tokens can represent data or control flow. A predicate is associated with each edge to regulate token flow.

Nodes in an SPG can also be treated as black boxes that perform some arbitrary computation. This property is essential for representing programs in languages such as Prolog [38,86], that cannot be easily represented using the default notion of a node. Annotations can also be attached to nodes to record arbitrary properties [71]. Nodes can be grouped to more easily allow annotations to refer to sets of nodes.

SPG's are designed to allow the computation of control-flow views, dataflow views, execution views, and "abstract views of system structures"[§§§§§§]. In addition, SPG's are designed to allow views to be automatically updated every time a change is made to one of them. ViewForm would likely to benefit from an internal representation that can represent as wide a diversity of programming languages as SPG's. Viewcode, however, must remain as comprehensible users in order to facilitate user interaction.

[§§§§§§] Abstract views of system structures are ways of viewing the overall system from a higher-level point of view.

6.1.4 Documenting Programs Through Views

The Chapter and Verse Program Description [89] is a method of documenting a program from a variety of views. The views are organized hierarchically and vertically, from a high-level overview of the program's purpose to the lower-level implementation details. Unlike ViewForm, the Chapter and Verse system is a methodology and does not provide automation. For example, it does not provide any automation for recomputing the documentation when changes are made to the program. Since Chapter and Verse is not automated, it would not necessarily preclude future programmers from having to pay some price for broken modularity.

6.2 Special-Purpose Languages via Transformations

In several systems, special-purpose or higher-level languages are "created" by virtue of a transformation system that translates the new languages to a base concrete language. These systems assume that the source program is complete and correct, and then apply a set of correctness-preserving transformations to "implement" the source program in the target language. The source and target languages may or may not be similar. These systems described below demonstrate different ways of building transformation systems. For each system, its approach to transforming programs is compared to ViewForm's approach.

6.2.1 TXL

TXL, the Turing eXtender Language [17], is a transformation system designed to easily allow new language constructs to be added to an existing base language. It has been used to provide object-oriented extensions to a language [18] and to automatically generate the code necessary to call a C graphics library from Prolog [19]. In addition, TXL has an associated denotational semantics [63]. Like ViewForm, TXL system contains a parser, a transformation engine, and an unparser. The syntax of the source language is described in a notation similar to BNF. Also similar to ViewForm, the parser takes a source program, converts it into a typed tree, and then gives it to the TXL transformation engine. The transformations are recursive, context-sensitive tree rewrite rules that can call arbitrary external procedures, much like the ViewForm walker view. Unlike ViewForm, however, a rewrite rule can only replace nodes in the subtree it is given. The engine applies a single, main rule to the tree. This rule is a composition of user-defined subrules. After the main rule has been applied, the unparser can be invoked to output the target program.

While TXL is remarkably similar to ViewForm in many ways, it differs in at least one key aspect: a rewrite rule cannot modify or view any code not in its given subtree. This implies that TXL does not allow non-local program analysis to be arbitrarily performed on specific pieces of

code. This will render TXL ineffective on the simulator and pedigree examples, which require access to non-local information to verify preconditions and to modify the uncoupled program.

6.2.2 ASCENT

ASCENT [37] is a system for building special-purpose languages out of general-purpose languages. ASCENT assumes the existence of a base programming language and environment. The new language is built by modifying the set of production rules that describe the base language's syntax. ASCENT then partially automates the production of a set of transformation rules that map constructs from the new language to the original base language. As the program is being transformed, ASCENT associates the new, target-language program expressions with the source program expressions from which they were derived. Unlike ViewForm, ASCENT does not provide any form of non-local program information that can be used in the production rules. This precludes the application of ASCENT on examples such as the simulator and pedigree examples which need non-local program information.

6.2.3 Proxac

Proxac [91] is a user interface for applying transformations to a source program. It displays the system-provided transformations and the program they are being applied to. The user selects the next transformation to be applied and can then see the results. While Proxac can be a useful tool for testing individual rewrite rules, it requires a high degree of user interaction and assumes that the user can evaluate the transformation results at each step along the way. This approach is too user intensive for ViewForm, since ViewForm already validates program modifications using predicates.

6.2.4 Elaborations

Elaborations [29] are a mechanism for developing programs top down, starting with a base high-level description. Elaborations are progressively lower-level descriptions (i.e., implementations) of the higher-level descriptions. Elaborations also provide a single vertical and hierarchical view of a program, and do not provide composable explanations for rewrite rules. The limitation of a single view on a program limits the kinds of preconditions that can be expressed, making Elaborations unsuitable for the simulator example (where a new view was required).

6.2.5 Darlington's User-Interactive Transformation System

In the transformation system described by Darlington [21], a clearly written source program is transformed into a more efficient program (in the source language) via a set of transformations. While the system is shown to successfully transform some small programs, the system re-

lies heavily on user interaction for deciding which transformations to apply. For example, the user is shown a program that has been transformed and is asked whether it is adequate. This presents a significant difficulty to a user who may not be able to recognize whether a certain transformation has produced an effect that will lead to the desired target program or the desired performance. ViewForm, on the other hand, strives to be as automated as possible. One can imagine ViewForm as an advance on Darlington's work if one considers predicates to be "automated users" that provide the kind of feedback Darlington's system depends upon.

6.2.6 PECOS and LIBRA

PECOS [4] is a knowledge-based system that takes a high-level specification and implements it by transforming it into a concrete program. PECOS draws transformations from a database of about 400 rules of which 75% are not directly related to the target language. When several rules are applicable, PECOS can call on LIBRA [50], an external program that estimates which of the applicable rules will lead to a more efficient program. The system can also interact with the user for assistance. This system is similar to ViewForm in that it provides a mechanism for reducing the amount of user interaction by relying on an external program designed to make reasonable decisions. PECOS, however does not support multiple views, making it unsuitable for the pedigree and simulator examples.

6.2.7 Cheatham's Transformation System for Reuse

Cheatham proposes a transformation system that supports the reuse of abstract programs [13]. The source programs are written in EL1, a high-level language with the means for syntactic extension. The source program is transformed into a lower-level concrete language by sets of transformations and a control structure. The transformations are pattern based, but like ViewForm, a semantic precondition can be associated with each pattern. The transformation system performs a limited amount of semantic analysis of the source program. Cheatham describes two interesting forms of reuse: *Rapid Prototyping* and *Custom Tailoring*. The former involves the rapid development of a working source program without significant regard for performance. The latter is the derivation of an efficient concrete program from a piece of working source code. Cheatham's system does not support arbitrary user-defined views, making it unsuitable for the pedigree and ViewForm examples.

6.2.8 CIP

The CIP system [5] was designed to investigate the construction of concrete programs from a formal specification via a transformation library. The source language is CIP-L, a wide-spectrum language that is extensible via schemas. The developers of CIP claim that user interaction, espe-

cially for higher-level decisions, will likely be necessary for large, non-toy applications. They do concede, however, that straightforward, mechanical, lower-level tasks could be done automatically. CIP, however, supports only correctness-preserving transformations and does not support multiple views of the source program. This makes it unsuitable for either the pedigree or simulator examples, which require non semantics-preserving transformations and multiple views.

6.3 Open Implementations

A more recent approach for building implementation-coupling systems is to use an *open implementation* [52,69,70] approach. Specifically, substantial work has been done with Metaobject Protocols (MOPs) [55,54]. A MOP opens an implementation by allowing a user to control important aspects of an implementation's decision-making process. This control is provided by documenting a set of metaobjects and a protocol that the implementation adheres to. A metaobject is an object that controls some aspect of an implementation's behavior, usually by making a decision. The protocol describes the interaction between the metaobjects themselves and between the metaobjects and other parts of the implementation. MOPs are designed to provide the user with both locality and incrementality, among other benefits.

I did not pursue a MOP-based approach for writing a view-based abstraction implementation because a MOP-based approach would require annotations on the source code. This would not be backwards compatible regardless of the source language, since it would require modifying the original source code. Nevertheless, as was discussed in Chapter 2, the MOP-based work below greatly influenced of view-based abstraction's design and implementation.

6.3.1 Intrigue

Intrigue [59] is a MOP for a Scheme compiler. The Intrigue architecture consists of two parts; a general data-flow engine (see Section 6.3.3) and an abstract data structure capable of representing Intrigue's intermediate languages. Intrigue's MOP allows users to control the compilation process via annotations on the source program, via user-defined object classes that localize and represent interesting program elements, and via user-defined methods that override or amend the compiler's default actions on those program elements. Intrigue's data-flow engine performs non-local analyses on its input program and provides support for user-defined, iterative data flows. ViewForm and view-based abstraction differ from Intrigue in several different aspects. Primarily, they are not compilers, and they do not require annotations on the source code. ViewForm can also work on separate contexts, whereas Intrigue does not support separate data-flow information for separate pieces of code. ViewForm's views are also specific, whereas Intrigue's data-flow engine is quite general. This difference allows ViewForm's views to be com-

puted more quickly, thus enhancing its practicality. If view-based abstraction were allowed to use annotations and performance was not a serious concern, Intrigue could be used as a basis for implementing a version of view-based abstraction with capabilities similar to ViewForm's.

6.3.2 Anibus

Anibus [76,77] is a MOP for controlling the coarse-grained parallelization of a program. Anibus's source language is sequential and its target language is explicitly parallel. A source language program is parallelized via annotations on program expressions. The annotations specify the program's parallelization and are used by Anibus to transform the program into the target language. A user can add new parallelizing transformations or modify existing ones by using Anibus's MOP. Anibus's transformation system is divided into three phases that implement the distribution of the program, the distribution of the data, and synchronization. The Anibus architecture without a MOP can be easily implemented as couplers with ViewForm.

6.3.3 Data Path Macros

Data Path Macros (DPMs)[56] are macros that may expand in arbitrary parts of a program. The DPM engine takes a source program and converts it into a program tree. In this tree, the nodes are objects representing syntactic program elements and the edges are objects representing the ways in which those elements are connected via s-expressions. This representation is similar to viewcode, especially since it provides up linking.

A DPM is a set of generic functions on objects in the program tree. These generic functions have access to data-flow information computed by the DPM engine. Syntactic annotations on the source program denote the places where a DPM is to be applied. The annotations make the corresponding program tree nodes easily recognizable to specific DPMs.

A DPM is created by writing a set of generic functions that perform transformations on the program tree and by defining a set of objects that will be used to annotate program tree nodes. One important property of DPMs is the ability to specify new user-defined data-flow computations. Such a specification describes initial data-flow values, how values propagate throughout the program tree, how values are combined at join points, and how equality between flow values is determined. This specification is procedural and takes the form of generic function methods. By using object-oriented inheritance, the amount of code that must be written to define a new DPM or a new data flow is significantly reduced. As discussed in Section 2.4.2, some of the notions underlying ViewForm and view-based abstraction have their roots in DPMs. Three of these notions are the utility of arbitrary non-local program analyses, of non semantics-preserving modifications, and of extensibility in a transformation system. In addition, although ViewForm is

not object oriented, an efficient generic dispatch system [58] would certainly be quite useful in its implementation.

6.4 Optimizing Programs via Transformations

Several approaches for building or designing optimizers are based on transformation systems. These systems are concerned with high-level source programs and low-level target programs. These systems deal with the same kinds of issues faced by ViewForm and view-based abstraction, though they are more specific.

6.4.1 Dora

Dora [26,25] is a transformation system for experimenting with compiler optimizations. Dora's intermediate language (IL) is DILS. DILS is based on Scheme and is extensible by virtue of being a schema. Instead of employing a traditional attribute grammar, Dora uses *attribution pattern sets* [27] to provide a more flexible framework for attributing the nodes of the IL tree. The same tree-pattern language is used to express both the attribution equations and the transformation pattern-matching language. This tree-pattern language is implemented via an efficient automata-based algorithm [43,10]. In Dora, the way in which transformations are applied can be specified from a fixed set of options. For example, the IL tree can be traversed from top to bottom, left to right, or the reverse of either option. Transformations can also be applied once or repeatedly. Another option is whether a transform's applicable sites should be recomputed every time a change is made to the IL tree or whether all of the applicable sites should be computed once, before the transformed is applied.

Dora transforms have access to non-local program information and can call arbitrary Scheme forms when a pattern is matched. While this architecture is similar to ViewForm's, the non-local attributes in Dora are computed automatically from a set of attribute definitions. This can lead to performance problems in the computation of non-local program information because the information is computed globally instead of on specific contexts as is done in ViewForm.

6.4.2 GENesis

GENesis [95] is an optimizer generator whose specification language is GOSpeL. GENesis takes a GOSpeL specification and automatically generates an optimizer. User interaction can be part of a GOSpeL specification in cases where several optimizing transformations are applicable. An optimizing transformation consists of a pattern, an enabling condition, and an action. The pattern is syntactic and the enabling condition can be in terms of non-local semantic program properties. When the pattern is matched and the precondition is met, the corresponding action

is invoked. GOSpeL's action language consists of five primitives that allow program elements to be deleted, copied, moved, added (i.e., synthesized), and modified. While this architecture bears resemblance to that of ViewForm, GENesis patterns are not arbitrarily general in that they cannot incorporate arbitrary pieces of non-local code. This means that GENesis cannot be used to express implementation couplings such as what is done for the simulator example.

6.4.3 Program Optimization and Derivation via Transformations

In [8], transformations are investigated as a form of optimizing or deriving programs by transforming and connecting axioms. One interesting aspect of this work is that transformations can assert information about parts of the program that they create. For example, a transformation that reifies an implicit loop can assert a type and numerical range on the loop variable. These kinds of assertions are consistent with ViewForm's design, since one of ViewForm's aims is to support interaction with entities that can provide more accurate and more available knowledge about pieces of the program. While ViewForm is not axiom-based, experimenting with axiom-based views is a viable direction for studying user interaction.

6.5 Interactive Program Design and Construction

A few transformational systems are designed with the end goal of assisting users in the design and construction of software. As described in [75], the vision discussed in [32] is one of the earliest examples of a Programmer's Assistant. These systems are generally related to ViewForm in that ViewForm permits user interaction.

6.5.1 KBEmacs

KBEmacs [93] the Knowledge-Based Editor in Emacs, is an editor that partially acts as a Programmer's Apprentice [74] during the construction of a program. It uses a knowledge base of *clichés*. A cliché is an algorithmic fragment about which both KBEmacs and the programmer have knowledge. KBEmacs is designed to complement a software developer's skills by automatically handling certain details of a program development such as checking for errors, filling in pieces of source code, and rearranging the code when necessary. While KBEmacs was not designed to solve the implementation-coupling problem, its use of clichés could be beneficial in providing a useful interface for user interaction in ViewForm.

6.5.2 A Program Verifier Assistant

The Designer/Verifier's Assistant [67] is a system designed to help programmers verify the designs of their programs. The user gives the system a specification of the verification conditions for a program. The system uses this specification to verify a program, to verify changes to a pro-

gram, to provide explanations for proposed changes, and to answer queries from the user. For example, the system can speculate on the effects that a proposed change will have with respect to a program's verification specification. The Assistant does not, however, support arbitrary non-local program analyses. This differentiates it from ViewForm, although its specification language is an excellent candidate for an alternative to the English-language ViewForm explanations. This could allow, for example, action explanations to serve as specifications for cross-checking the intent of a program modification.

6.5.3 CCEL

CCEL [23,14] is a language for describing user-defined constraints on a C++ program. A user can describe design, implementation, and stylistic constraints. When a program is compared to its CCEL specification, any inconsistencies are reported via default system error handlers. The default method of reporting an inconsistency is to display the constraint that was violated. A user can optionally associate a more descriptive message that is displayed when a constraint is violated. While CCEL is specific to C++ and cannot perform transformations on the source program, it can be used (as above) to specify a different sort of action explanation. This kind of explanation could be used as a debugging tool to ensure that an action's program modification satisfies a set of constraints.

6.6 Future Work

View-based abstraction and ViewForm can be extended in various ways. Some interesting directions are discussed below.

6.6.1 Reusability

Automatically extracting reusable program slices from poorly structured code is an active area of research [60]. View-based abstraction can be used as a model for reasoning about such slices. A simple example of this is the inlining performed for the pedigree example. While the criterion for this example was solely performance, the process for extracting a slice containing `sc-*` would be very similar. For example, extracting a program slice using view-based abstraction would involve writing predicates that validate the slice as a piece of code that can be substituted into another context. This is exactly what happened in the function inlining aspect of the pedigree example, except that the ViewForm code was mostly specific to the pedigree code. A more general piece of slicing code would be interesting to pursue.

A program slicing [94] ViewForm view would also be interesting to pursue for this approach.

6.6.2 Symmetry in Couplings

View-based abstraction is asymmetric in that it manages implementation couplings but not interface couplings. It would be interesting to allow view-based abstraction to handle both. One problem this could be used to explore is portability. For example, suppose that a programming language's interface requires certain core interface functions and makes others optional (as is the case with Scheme). A program that uses anything beyond the core interface functions is therefore not automatically portable to an implementation that only implements the code interface.

If the interface itself was part of a coupling managed by view-based abstraction, however, this loss of portability might not be necessary. For example, when a program is run under a new language implementation, a view could collect uses of the optional functionality (i.e., `make-rectangular`). For each optional piece of functionality found, a predicate could determine whether the functionality is actually implemented and available. This is the step that would be mediated by an interface view on the language implementation, which in the simplest case would look for functions with names given in the language specification. If the functionality was implemented, the next candidate would be tested. Otherwise, an action would be invoked that would insert code that implements the functionality into the context. This code would be written using the core functionality provided by every language implementation. If successful, this approach would not only increase portability, but it would also allow programs to run as efficiently as the underlying, native functionality permits.

This scenario assumes that every piece of optional functionality can be expressed by the core functionality. The circumstances under which this may or may not be possible can be examined by future research efforts into this problem.

6.6.3 Other Programming Paradigms

A hybrid approach between Stata's work [84] and view-based abstraction would be interesting to pursue. One can imagine view-based abstraction being used to enhance the maintainability of the code upon which Stata's specialization interfaces are based (thus giving programmer more implementation freedom). For example, view-based abstraction could be used to express implementation specializations, and the specialization interface specifications could be used to automatically validate the resulting modifications.

Another piece of future work is developing a C[51] based or Java[39] based view-based abstraction implementation. C is a widely used language, and would give view-based abstraction a huge variety of opportunities for improvement. Java's newness would provide an opportunity for view-based abstraction to enter early into its programming environment's development, giving

view-based abstraction a better chance at becoming prevalent. By its nature, Java would also provide a test bed for exploring distributed views and distributed couplings.

For these languages, view computations could be more difficult due to the existence of raw pointers and pointer arithmetic in C and inheritance in Java. As previously discussed, the ability to compute useful view information is one of a view-based abstraction implementation's more interesting aspects.

6.6.4 An Experiment

While the examples presented in Chapter 5 are thousands of lines long or smaller, real-world programs can be hundreds of thousands or millions of lines long[3] and can undergo decades of maintenance.[*******] A true experiment for testing view-based abstraction under these real-world conditions would therefore be quite useful. Of particular interest is studying how couplings within the code evolve over time, what kinds of views are deemed useful or necessary, and what kinds of library functions are most commonly used.

[*******] The "year 2000" problem has made it clear that many programs survive for well longer than originally designed.

Chapter 7

Conclusion

7.1 Summary

Maintainability and modularity in the presence of implementation coupling are not possible under black-box abstraction. To approach this problem, I developed a six-step model of implementation coupling. From this model, I gained two critical ideas. The first is that modularity is lost only if an implementation coupling persists through a modification that invalidates it. The second is that the implementation-coupling process (including the recoupling step) can be automated by providing a language for expressing couplings. These ideas form the main thrust in this dissertation: demonstrating how to automate the six-step process in a practical way, thereby solving the implementation-coupling problem. This thrust evolves into the development of view-based abstraction, an easy to understand, language independent, backwards compatible, incremental, and time-saving abstraction model.

View-based abstraction introduces a model and methodology for performing implementation couplings. The model contains components for reasoning about implementation couplings, and the methodology provides a structured way for using the components to express implementation couplings via the six-step process. The view-based abstraction components include contexts for representing modules, predicates for expressing preconditions, actions for expressing program modifications, dispatchers for combining predicates and actions, rules for combining the constructs in general, couplers for automatically generating couplings, and views for determining non-local program properties. Most significantly, the model requires a view invalidation/recoupler, responsible for automatically carrying out the recoupling step. Making the view invalidation/recoupler work is both necessary and sufficient for solving the implementation-coupling problem.

I also developed ViewForm, an implementation of view-based abstraction. ViewForm uses source-to-source transformations as the basic coupling language paradigm. ViewForm introduces

the vform, a novel construct used to implement every view-based abstraction construct. ViewForm also implements contexts, three default views, a higher-order view, a small but effective library of higher-level coupling construct constructors, and an effective view invalidation/recoupler.

To demonstrate view-based abstraction, I used ViewForm to perform implementation couplings on three examples: an amorphous computing simulation, a pedigree example, and ViewForm itself. I did not write either the simulator or the pedigree example code. In addition, I did not design the simulator coupling and one of the pedigree couplings. This makes the simulator a completely independent example, and the pedigree example a mostly independent example. The three examples demonstrate how implementation couplings can be made to maintain modularity. In addition, the examples demonstrate how view-based abstraction is a language independent, understandable, backwards compatible, incremental, and time-saving abstraction model.

The examples further demonstrate the pragmatics of view-based abstraction. Even though ViewForm analyzed the example code with a LIAR-style interprocedural data-flow analysis, the ViewForm code ran in 33% to 40% of the time needed to compile the example code using the MIT Scheme compiler.

7.2 Contributions

This dissertation's central goal is a solution to black-box abstraction's implementation coupling problem. Towards this goal, this dissertation makes a variety of contributions that have immediate impact on software design and maintainability. In the long term, this dissertation can affect programming environment design, language design, and transformation system design.

One up-front contribution is a vocabulary for discussing the implementation-coupling problem and for making distinctions among its aspects. These vocabulary terms include implementation coupling, interface coupling, invalidating implementation, coupled code, uncoupled code, coupling code, views, contexts, and complexity layering. In the Chapter 6 literature survey, I did not find explicit or mutually consistent names for these notions.

This dissertation's foundational contribution is view-based abstraction, a model for eliminating the modularity problems associated with implementation coupling. By providing programmers with a way to make a coupling's normally implicit preconditions and modifications explicit and imperative, programmers can shift the burden of maintaining the coupling to the computer. View-based abstraction is also backwards compatible, language independent, and incremental. This leads to another contribution: view-based abstraction can be applied to existing code im-

mediately following the availability of a view-based abstraction implementation for the code's programming language. This can happen even if full access to the source code is not available.

I contribute to the design of programming environments with the notions of non-local but practical views specific to contexts. Even without view-based abstraction, these kinds of views and contexts are useful programming tools that can be included in existing and future programming environments. They provide programmers with accurately computed non-local program information that otherwise can and often does contain errors when computed manually. I contribute with guidance in the kinds of views that I have found useful, such as naming, data-flow, and expression typing.

This dissertation's contributions also include ViewForm, a full view-based abstraction implementation. ViewForm contributes to the field of program transformation systems by demonstrating how to build a transformation language that provides support for computing non-local program properties across abstraction boundaries, performing non semantics-preserving transformations, and accepting user interaction. To my knowledge, no current transformation language provides support for all three (for a variety of performance or semantic reasons). I show that the implementation coupling problem provides ample motivation for why transformation-language support for these features is useful, and ViewForm shows how it can be done.

ViewForm also shows how the view-based abstraction specification can be implemented to reduce the burden of complexity on its users. In particular, I demonstrate how a complexity layering approach can provide a measure of simplicity for common uses while still permitting more powerful operations at other complexity layers. This contribution can influence the implementors of program transformation systems to adopt similar strategies for making common transformations easier to express.

As part of ViewForm's implementation, I show how vforms can be implemented, and how views, predicates, actions, dispatchers, rules, and couplers can be implemented from vforms. Since these constructs functionally correspond to many commonly found in related program transformation systems, they suggest the existence of a simple, operational, and compositional model for generally describing program transformation systems.

This dissertation demonstrates how view-based abstraction and its implementation can be non-pervasively added to a programming environment. Given the backwards compatible and incremental nature of view-based abstraction, this suggests a low cost for including view-based abstraction in future programming environment implementations. I also provide guidance and evidence of which views and coupling construct constructors are useful to include by default in a library. The three coupling examples contribute with guidance to programmers developing their own implementation couplings. In addition, this dissertation provides direction on how

programmers can better manage complexity by decreasing the overall amount of coupling between modules without necessarily losing performance.

7.3 Conclusion

This dissertation has demonstrated that the coexistence of implementation coupling, modularity, maintainability, and true abstraction is possible. By focusing on the process by which a programmer might otherwise couple implementations, I gained insight into the distinct steps of the process. This insight allowed me to distinguish between the computable and non-computable steps, leading to the development of view-based abstraction. As for now, there are many existing circumstances that can benefit from view-based abstraction, as well as many directions to further explore its use.

Appendix A

Selected Aspects of the ViewForm Interface

A.1 ViewForm Expression Type Testers

These type testers take a viewcode expression and return #t if the expression is of a given type.

```
type/=>?                    type/define-macro?          type/number?
type/access?                type/delay?                 type/or?
type/and?                   type/do?                    type/proc-maker?
type/begin?                 type/fixnum?                type/quasiquote?
type/bool?                  type/fluid-let?             type/quote?
type/case?                  type/if?                    type/rational?
type/char?                  type/inexact?               type/set!?
type/complex?               type/integer?               type/string?
type/cons-stream?           type/lambda?                type/the-environment?
type/cond?                  type/let?                   type/unquote?
type/declare?               type/let*?                  type/unquote-splicing?
type/default-object??       type/letrec?                type/unsyntax?
type/define?                type/load-option?           type/var-ref?
type/define-integrable?     type/named-lambda?          type/vector?
type/define-structure?      type/named-let?
```

A.2 Viewcode Expression Return Types

These return types are computed by the liar view for each viewcode expression.

```
*any-type*                  *complex-type*              *symbol-type*
*unspecified*               *pair-type*                 *null-type*
*number-type*               *list-type*                 *env-type*
*integer-type*              *fresh-list-type*           *bottom-type*
*inexact-type*              *vector-type*               *stream-type*
*rat-type*                  *string-type*               *hash-table-type*
*fix-type*                  *character-type*            *bit-string-type*
*non-negative-fix-type*     *proc-type*                 *weak-pair-type*
*non-negative-int-type*     *bool-type*                 *pathname-type*
```

A.3 Miscellaneous ViewForm Functions

(top-level-exp->origin exp context)

Returns the top-level expression `exp`'s origin in `context`, or `#f` if exp is not a top-level expression in `context`

(exp->all-procs exp context)

Returns a list of all viewcode expression that can create a procedural return value for `exp` in `context`

(top-lvl-exp->free-vars exp context)

Returns a list of the free variables in the top-level expression `exp` in `context`

(fully-qualify dir filename)

`dir` and `filename` are strings representing a directory and filename. Returns a fully qualified pathname for use by `make-context-from-file` or `make-context-from-files`

(consumer->primitive-producers exp context)

Returns a list of all viewcode expressions in `context` that can produce a value for `exp`. These expressions include constants, some predefined primitive procedures, and some special forms.

A.4 Viewcode Canonicalization

ViewForm makes three changes to Scheme code when it converts it into viewcode. These are:

- All define expressions are converted to the form: `(define <var> <body>)`
- All `define`, `lambda`, named `lambda`, `let`, `let*`, `letrec`, named `let`, and `fluid-let` bodies are converted to `begin` expressions if the body contains more than one expression
- Any conversions performed by the MIT Scheme reader

Appendix B

Implementation of Selected ViewForm Functions

```
(define (top-lvl-name->def var-name context)
  (let ((creation (top-level-defined? var-name context)))
    (if creation
        (up-link-n creation 2) ;up link to (Define... var-name ...)
        (error "Var name not defined: " var-name))))

(define-integrable (make-predicate/check-ret-type type expl-string)
  (make-explained-predicate
   (lambda (vform context vc-exps)
     (for-all? vc-exps
       (lambda (vc-exp)
         (for-all? (consumer->primitive-producers vc-exp context)
           (lambda (producer)
             (types-consistent? type (exp->return-types producer context)))))))
   expl-string))

(define (make-predicate/check-proc-param-types type-specs)
  (make-explained-predicate
   (lambda (vform context vc-exps)
     (for-all? vc-exps
               (lambda (vc-exp)
                 (for-all? (consumer->primitive-producers vc-exp context)
                           (lambda (producer)
                             (and (type/proc-maker? producer)
                                  (for-all? type-specs
                                            (lambda (type-spec)
                                              (let ((selector  (car type-spec))
                                                    (predicate (cdr type-spec)))
                                                (do-vform predicate
                                                          context
                                                          (list (selector
                                                            (lambda/formals producer)
                                                                  ))))))))))))
   "exp's returning procedures whose parameters pass given predicates"))
```

```
(define (make-predicate/find-declaration decl-search)
  (make-explained-predicate
   (lambda (vform context vc-exps)
     (list-search-positive (context/exps context)
       (lambda (exp)
         (and (type/declare? exp)
              (there-exists? (declare/body exp)
                (lambda (decl-list)
                  (there-exists? decl-list
                    (lambda (decl)
                      (cond ((vc-enc? decl)
                             (vc-eq? decl decl-search))
                            ((pair?   decl)
                             (vc-eq? (car decl) decl-search))
                            (else (error "Unknown Declaration Syntax: " decl))))))))))
   (string-append " " (symbol->string decl-search) " declaration")))

(define (make-simple-predicate vc-exp-pred exp-type)
  (make-explained-predicate (lambda (vform context vc-exps)
                              (vc-exp-pred (car vc-exps)))
                            exp-type))

(define (make-predicate/exp-type pred exp-type)
  (make-simple-predicate pred (string-append " " exp-type)))

(define *conjunctive-1-to-1-control-structure*
  (lambda (vforms)
    (lambda (vform context vc-exps)
      (for-all-lists? (lambda (vform vc-exp) (do-vform vform context (list vc-exp)))
                      vforms
                      vc-exps))))

(define (combine-vforms-conjunctive-1-to-1 . vforms)
  (combine-vforms *conjunctive-1-to-1-control-structure*
                  *id-expl-combiner*
                  vforms))

(define *do-all-vform-control-structure*
  (lambda (vforms)
    (lambda (vform context vc-exps)
      (let outer-loop ((vc-exps vc-exps)
                       (result  #t))
        (if (null? vc-exps)
            result
            (let ((vc-exp-list (list (car vc-exps))))
              (outer-loop (cdr vc-exps)
                          (let inner-loop ((the-vforms vforms))
                            (if (null? the-vforms)
                                #f
                                (begin (delegate-vform (car the-vforms)
                                                       context
                                                       vc-exp-list
                                                       vform)
                                       (inner-loop (cdr the-vforms))))))))))))

(define (combine-vforms-do-all . vforms)
  (combine-vforms *do-all-vform-control-structure*
                  *id-expl-combiner*
                  vforms))
```

```
(define (for-all-lists? predicate list1 . lists-n)
  (let loop ((lists (cons list1 lists-n))
             (result #t))
    (if (null? (car lists))
        result
        (loop (map cdr lists)
              (and result (apply predicate (map car lists)))))))

(define (combine-vforms-conjunctive . vforms)
  (combine-vforms *conjunctive-control-structure*
                  *id-expl-combiner*
                  vforms))


(define *and-all-vform-control-structure*
  (lambda (sub-vforms)
    (lambda (vform context vc-exps)
      (for-all? sub-vforms
        (lambda (sub-vform) (do-vform sub-vform context vc-exps))))))

(define *conjunctive-control-structure* *and-all-vform-control-structure*)

(define (make-explained-predicate predicate exp-type)
  (make-predicate predicate
                  (make-expl "for identifying" exp-type)))

(define (make-simple-action simple-modifier expl-string)
  (make-action (lambda (vform context vc-exps)
                 (simple-modifier vform context (car vc-exps)))
               (make-expl "for" expl-string)))

(define (make-simple-exp-action modifier expl-string)
  (make-action (lambda (vform context vc-exps) (apply modifier vc-exps))
               (make-expl "for" expl-string)))

(define (make-explained-action action expl-string)
  (make-action action (make-expl "for" expl-string)))

(define (make-simple-dispatcher predicate action)
  (make-dispatcher predicate
                   action
                   *default-expl-combiner*))

(define *default-expl-combiner*
  (lambda (vforms)
    (lambda (vform context vc-exps)
      (format #t "~S containing " (vform/identifier vform))
      (for-each (lambda (vform)
                  (explain! vform context vc-exps))
                vforms))))

(define *id-expl-combiner*
  (lambda (vforms)                      ;default just runs the explanations
    (lambda (vform context vc-exps)
      vform                             ;ignore
      (newline)
      (for-each (lambda (vform)
                  (explain! vform context vc-exps)
                  (newline))
                vforms))))
```

```
(define (verbose-expl-combiner default-expl)
  (lambda (vforms)
    (let ((verbose-expl (*id-expl-combiner* vforms)))
      (lambda (vform context vc-exps)
        (let ((vform-type (vform/identifier vform)))
          (format #t "~%~S for ~S " vform-type default-expl)
          (if *verbose*
              (begin
                (format "Containing: ~%")
                (verbose-expl vform context vc-exps))))))))

(define (var-bound-by? var binding context)
  (eq? (var-ref->binding var context) binding))

(define (quoted-symbol? exp)
  (and (type/quote? exp)
       (vc-symbol? (quote/thing-quoted exp))))

(define (only-calls? symbol vc-exp context)
  (let ((binding (top-level-defined? symbol context)))
    (cond (binding (%only-calls? (top-lvl-name->def symbol context) vc-exp context))
          ((lookup-predefined symbol) (%only-calls-primitive (lookup-predefined symbol)
                                                             vc-exp
                                                             context))
          (else (error "[only-calls?] Don't know how to handle: " symbol)))))

(define (%only-calls? binding vc-exp context)
  (let ((def-prods    (consumer->primitive-producers (lambda/body (define-var/value binding))
                                                     context))
        (vc-exp-prods (consumer->primitive-producers vc-exp context)))
    (and (for-all? vc-exp-prods
           (lambda (producer)
             (memq producer def-prods)))
         (for-all? def-prods
           (lambda (producer)
             (memq producer vc-exp-prods))))))

(define (%only-calls-primitive prim-binding vc-exp context)
  (let ((vc-exp-prods (consumer->primitive-producers vc-exp context)))
    (for-all? (append-map (lambda (prod)
                            (append (exp->consts   prod context)
                                    (exp->primops  prod context)
                                    (exp->sp-forms prod context)))
                          vc-exp-prods)
      (lambda (producer)
        (eq? producer prim-binding)))))

(define (make-simple-expl text)
  (lambda (vform context vc-exps)
    (format #f "~%~S ~S" (vform/identifier vform) text)))

(define-integrable (make-expl text exp-type)
  (make-simple-expl (string-append text " " exp-type " expressions")))
```

```
(define (y-or-n-user-response)
  (format #t " ~%Shall I accept this and continue? ~%Enter y to accept or n to reject: ")
  (let ((user-response (read)))
    (cond ((or (eq? user-response 'y) (eq? user-response 'yes)))
          ((or (eq? user-response 'n) (eq? user-response 'no)) #f)
          (else (format #t "~%Unrecognized Response, please try again")
                (y-or-n-user-response)))))
```

Bibliography

[1] **Harold Abelson, Gerald Jay Sussman, and Julie Sussman** *Structure and Interpretation of Computer Programs* Second Edition, MIT Press, Cambridge, MA, 1996.

[2] **Mehmet Aksit, Lodewijk Bergmans, and Sinan Vural** *An Object-Oriented Language-Database Integration Mode: The Composition-Filters Approach* In *European Conference on Object-Oriented Programming*, O. Lehrmann Madsen Ed., Lecture Notes in Computer Science 615, Utrecht, The Netherlands, June/July 1992.

[3] **Darren C. Atkinson and William G. Griswold** *The Design of Whole-Program Analysis Tools* In *Proceedings of the 18th International Conference on Software Engineering*, Pages 16-27, Berlin, Germany, March 1996.

[4] **David R. Barstow** *An Experiment in Knowledge-Based Automatic Programming* In *Artificial Intelligence* Volume 12, Pages 73-119, 1979.

[5] **Friedrich Ludwig Bauer, Bernhard Möller, Helmut Partsch, and Peter Pepper** *Formal Program Construction by Transformations—Computer Aided, Intuition-Guided Programming* In *IEEE Transactions on Software Engineering* 15(2), February 1989.

[6] **Michael R. Blair** *Descartes: A Dynamically Adaptive Compiler and Runtime System Using Continual Profile-Driven Multi-Specializations*, Ph.D. Proposal, MIT Laboratory for Computer Science and MIT Artificial Intelligence Lab, December 1992. ftp://ftp.swiss.ai.mit.edu/pub/users/ziggy/Papers/Thesis/PhD/PhD-proposal.ps

[7] **Michael R. Blair** *Descartes: Foundations in Dynamically Adaptive Source Compilation & Run-Time Execution Specialization*, Ph.D. Dissertation, MIT Laboratory for Computer Science and MIT Artificial Intelligence Lab, Forthcoming. http://www.swiss.ai.mit.edu/~ziggy/descartes.html

[8] **Manfred Broy and Peter Pepper** *Program Development as a Formal Activity* In *IEEE Transactions on Software Engineering* 7(1), Pages 14-22, January 1981.

[9] **Craig Chambers, David Ungar, and Elgin Lee** *An Efficient Implementation of SELF, a Dynamically-Typed Object-Oriented Language Based on Prototypes* In OOPSLA '89 Conference Proceedings, Sigplan Notices, **24**(10), 1989.

[10] **David R. Chase** *An Improvement to Bottom-Up Tree Pattern Matching* In *Conference Record of the Fourteenth ACM Symposium on Principles of Programming Languages* München, W. Germany, January 1987.

[11] **Marina Chen** *A Parallel Language and its Compilation to Multiprocessor Machines or VLSI*, In *Proceedings of the Thirteenth Symposium on Principles of Programming Languages* Pages 131-139, 1986.

[12] **Marina Chen, Young-il Choo and Jingke Li** *Compiling Parallel Programs by Optimizing Performance*, In *The Journal of Supercomputing* 2(2), Pages 171-207, October 1988.

[13] **Thomas E. Cheatham, Jr.** *Reusability Through Program Transformations* In *IEEE Transactions on Software Engineering* 10(5), Pages 589-594, September 1984.

[14] **Anir Chowdhury and Scott Meyers** *Facilitating Software Maintenance by Automated Detection of Constraint Violations* In *IEEE Conference on Software Maintenance* Montreal, Quebec, September 1993.

[15] **William Clinger and Jonathan Rees** *Macros that Work* In *ACM Symposium on Principles of Programming Languages*, Pages 155-162, 1991.

[16] **William Clinger and Jonathan Rees**, eds. *Revised^4 Report on the Algorithmic Language Scheme* http://swissnet.ai.mit.edu/scheme-home.html, November 1991.

[17] **James R. Cordy and Ian. H. Carmichael** *The TXL Programming Language Syntax and Informal Semantics: Version 7* Department of Computing and Information Science, Queens University at Kingston, Canada, June 1993.

[18] **James R. Cordy and Eric Promislow** *Specification and Automatic Prototype Implementation of Polymorphic Objects in TURING Using the TXL Dialect Processor* In *Proceedings of the IEEE International Conference on Computer Languages* New Orleans, March 1990.

[19] **James R. Cordy and Medha Shukla** *Practical Metaprogramming* External Technical Report 92-342, Department of Computing and Information Science, Queens University at Kingston, Canada, October 1992.

[20] **O.-J. Dahl, E. W. Dijkstra, C. A. R. Hoare** *Structured Programming* Academic Press, New York, NY, 1972.

[21] **John Darlington** *An Experimental Program Transformation and Synthesis System* In *Readings in Artificial Intelligence and Software Engineering* Charles Rich and Richard C. Waters, eds., Morgan Kaufmann Publishers, Inc., Los Altos, CA, 1986.

[22] **E. W. Dijkstra** *A Discipline of Programming* Prentice-Hall, Englewood Cliffs, NJ, Page 64, 1976.

[23] **Carolyn K. Duby, Scott Meyers, and Steven P. Reiss** *CCEL: A Metalanguage for C++* Technical Report CS-92-51, Brown University, October 1992.

[24] **Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa** *Demand-Driven Computation of Interprocedural Data-Flow* In *Proceedings of the 22nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* San Francisco, CA, Pages 37-48, January 1995.

[25] **Charles Donald Farnum** *Pattern-Based Languages for Prototyping of Compiler Optimizers* Technical Report Number UCB/CSD 90/608, University of California, Berkeley, December 1990.

[26] **Charles Farnum** *Dora - An Environment for Experimenting with Compiler Optimizers* In *Proceedings of the 1992 International Conference on Computer Languages* Oakland, CA, April 1992.

[27] **Charles Farnum** *Pattern-Based Tree Attribution* In *Proceedings of the Nineteenth Annual Symposium on Principles of Programming Languages* Albuquerque, New Mexico, January 1992.

[28] **Charles Farnum** *Personal Communication* April 1995.

[29] **Martin S. Feather** *Constructing Specifications by Combining Parallel Elaborations* Technical Report ISI/ES-88-216, Information Sciences Institute, University of Southern California, August 1988.

[30] **Stuart Feldman** *Make - A Program for Maintaining Computer Programs* In *Software - Practice and Experience*, 9(4), Pages 255-265, April 1979.

[31] **Robert W. Floyd** *Algorithm 97 (SHORTEST PATH)* Communications of the ACM, 5(6), Page 345, 1962.

[32] **Robert W. Floyd** *Toward Interactive Design of Correct Programs* IFIP, Ljubljana, Yugoslavia, Pages 7-11, August, 1971.

[33] **Bob French** *Personal Communication*, Hewlett Packard Company, April 3, 1997.

[34] **Keith Brian Gallagher and James R. Lyle** *Using Program Slicing in Software Maintenance* In *IEEE Transactions on Software Engineering* 17(8), Pages 751-761, August 1991.

[35] **Simson Garfinkel, Daniel Weise, and Steven Strassmann** *The UNIX-HATERS Handbook*, IDG Books Worldwide, San Mateo, CA, 1994.

[36] **David Garlan** *Views for Tools in Integrated Environments* Ph.D. Dissertation, Carnegie Mellow University, May 1987. Also appears as CMU Technical Report CMU-CS-87-147.

[37] **David Garlan, Linxi Cai, Robert L. Nord, and Robert Stockton** *ASCENT: Application-Specific Environment Transformer* Technical Report CMU-CS-92-180 Carnegie Mellon University, 1992.

[38] **F. Giannesini, H. Kanoui, R. Pasero, and M. van Caneghem** *Prolog* Addison-Wesley, 1986.

[39] **James Gosling, Bill Joy, and Guy Steele** *The Java Language Specification* Version 1.0, Addison-Wesley, 1996.

[40] **A.N. Habermann, Charles Krueger, Benjamin Pierce, Barbara Staudt, and John Wenn** *Programming with Views* Technical Report CMU-CS-87-177, Carnegie Mellon University, January 1988.

[41] **Mary W. Hall, Saman P. Amarasinghe, Brian R. Murphy, Shih-Wei Liao, and Monica Lam** *Detecting Coarse-Grain Parallelism Using an Interprocedural Parallelizing Computer* In *Proceedings of the 1995 ACM/IEEE Supercomputing Conference*, San Diego, California, December 1995.

[42] **Chris Hanson et. al.** *MIT Scheme Reference Manual* Edition 1.62 for Scheme Release 7.4, MIT AI Lab, April 1996.

[43] **C.M. Hoffman and M.J. O'Donnell** *Pattern Matching in Trees* In *Journal of the ACM* 29(1), pages 68-95, January 1982.

[44] **Yasuaki Honda and Mario Tokoro** *Soft Real-Time Programming through Reflection* In *Proceedings of the International Workshop on New Models for Software Architectures: Reflection and Meta-Level Architecture* Pages 12-23, Tokyo, Japan, November 1992.

[45] **John E. Hopcroft and Jeffrey D. Ullman** *Introduction to Automata Theory, Languages and Computation* Addison-Wesley Publishing Company, 1979.

[46] **Susan Horowitz, Jan Prins, and Thomas Reps** *Integrating Non-Interfering Versions of Programs* In *ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, San Diego, CA, Pages 133-145, January 1988.

[47] **Galen C. Hunt and Michael L. Scott** *Coign: Efficient Instrumentation for Inter-Component Communication Analysis* URCS Tech Report 648, February 1997.

[48] **W. L. Hürsch and Christina Lopes** *Separation of Concerns* Northeastern University Technical Report NU-CCS-95-03, February 1995.

[49] **IEEE Std 1178-1990** *IEEE Standard for the Scheme Programming Language* Institute of Electrical and Electronic Engineers, Inc., New York, NY, 1991.

[50] **Elaine Kent** *On the Efficient Synthesis of Efficient Programs* In *Artificial Intelligence* Volume 20, Pages 253-305, 1983.

[51] **Brian W. Kernighan and Dennis M. Ritchie** *The C Programming Language* Prentice-Hall, 1978.

[52] **Gregor Kiczales** *Towards a New Model of Abstraction in the Engineering of Software* In *Proceedings of the International Workshop on New Models for Software Architectures: Reflection and Meta-Level Architecture* Pages 1-11, Tokyo, Japan, November 1992. A revised version is also available at: http://www.xerox.com/PARC/spl/eca/oi/gregor-invite.html.

[53] **Gregor Kiczales** *Beyond the Black Box: Open Implementation*, IEEE Software, January 1996.

[54] **Gregor Kiczales, J. Michael Ashley, Luis H. Rodriguez Jr., Amin Vahdat, and Daniel G. Bobrow** *Metaobject Protocols: Why We Want Them, and What Else They Can Do* In *Object-Oriented Programming: The CLOS Prospective* Andreas Paepcke, Ed., MIT Press, Cambridge, MA, Pages 101-118, 1993.

[55] **Gregor Kiczales, Jim des Rivières and Daniel G. Bobrow** *The Art of the Metaobject Protocol*, MIT Press, Cambridge, MA, 1991.

[56] **Gregor Kiczales, John Lamping, Luis H. Rodriguez Jr., and Erik Ruf** *Macros that Reach Out and Touch Somewhere* Internal Technical Report, Embedded Computation Area, Xerox PARC, August 1992.

[57] **Gregor Kiczales, et. al.** *Aspect-Oriented Programming* Technical Report SPL97-008, Xerox PARC, February 1997.

[58] **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* Nice, France, Pages 99-105, 1990. Also In *Object-Oriented Programming: The CLOS Prospective*, Andreas Paepcke, Ed., MIT Press, Cambridge, MA, Pages 335-348, 1993.

[59] **John Lamping, Gregor Kiczales, Luis H. Rodriguez Jr., and Erik Ruf** *An Architecture for an Open Compiler* In *Proceedings of the International Workshop on New Models for Software Architectures: Reflection and Meta-Level Architecture* Pages 95-106, Tokyo, Japan, November 1992.

[60] **Filippo Lanubile and Giuseppe Visaggio** *Extracting Reusable Functions by Program Slicing* University of Maryland Computer Science Department Technical Report, CS-TR-3594, January 1996.

[61] **Barbara Liskov, Russell Atkinson, Toby Bloom, J. Eliot Moss, J. Craig Schaffert, Robert Scheifler, and Alan Snyder** *CLU Reference Manual* Springer-Verlag, 1984. Also published as Lecture Notes in Computer Science 114, G. Goos and J. Hartmanis, Eds., Springer-Verlag, 1981.

[62] **John M. Lucassen** *Types and Effects: Towards the Integration of Functional and Imperative Programming* Technical Report MIT/LCS/TR-408, MIT, August 1987.

[63] **Andrew Malton** *The Denotational Semantics of a Functional Tree-Manipulation Language* Computer Languages, 19(3), Pages 157-168, 1993.

[64] **Scott Meyers and Steven P. Reiss** *A System for Multiparadigm Development of Software Systems* Technical Report CS-91-50, Department of Computer Science, Brown University, August 1991.

[65] **Jim Miller** *Problem Set 9* Structure and Interpretation of Computer Programs, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Spring 1997.

[66] **Robin Milner, Mads Tofte, and Robert Harper** *The Definition of Standard ML*, MIT Press, Cambridge, MA, 1990.

[67] **Mark S. Moriconi** *A Designer/Verifier's Assistant* In *IEEE Transactions on Software Engineering* 5(4), Pages 387-401, July 1979.

[68] **Hideaki Okamura, Yutaka Ishikawa, and Mario Tokoro** *AL-1/D: A Distributed Programming System with Multi-Model Reflection Framework* In *Proceedings of the International Workshop on New Models for Software Architectures: Reflection and Meta-Level Architecture* Pages 36-47, Tokyo, Japan, November 1992.

[69] **Open Implementation** *OI Homepage* http://www.xerox.com/PARC/spl/eca/oi.html.

[70] **Open Implementation** *Record of the Workshop on Open Implementation*, Gleneden Beach, Oregon, October 1994. http://www.xerox.com/PARC/spl/eca/oi/workshop-94/default.html

[71] **Steven P. Reiss** *On the Use of Annotations for Integrating the Source in a Program Development Environment* Technical Report CS-91-32, Department of Computer Science, Brown University, April 1991.

[72] **Steven P. Reiss** *PECAN: Program Development Systems that Support Multiple Views* In *IEEE Transactions on Software Engineering* 11(3), March 1985.

[73] **Thomas Reps** *Algebraic Properties of Program Integration* In *Proceedings of the Third European Symposium on Programming* Pages 326-340, Copenhagen, Denmark, May, 1990.

[74] **Charles Rich and Richard C. Waters** *The Programmer's Apprentice* ACM Press, New York, NY, 1990.

[75] **Charles Rich and Richard C. Waters, Eds.** *Readings in Artificial Intelligence and Software Engineering*, Morgan Kaufmann Publishers, Inc., Los Altos, CA, 1986.

[76] **Luis H. Rodriguez Jr.** *Coarse-Grained Parallelism Using Meta-Object Protocols* S.M. Thesis, Massachusetts Institute of Technology, September 1991. Also Appears as Technical Report SSL-91-06, Xerox PARC, September 1991.

[77] **Luis H. Rodriguez Jr.** A *Study on the Viability of a Production-Quality Metaobject Protocol-Based Statically Parallelizing Compiler* In *Proceedings of the International Workshop on New Models for Software Architectures: Reflection and Meta-Level Architecture* Pages 107-112, Tokyo, Japan, November 1992.

[78] **Luis H. Rodriguez Jr.** *ViewForm: A Language for Building Program Transformation Systems* MIT AI Lab, Ph.D. Thesis Proposal, July 1995. *http://swissnet.ai.mit.edu/~lhr/papers/phd-prop/full-paper.html*

[79] **Guillermo J. Rozas** *LIAR: an ALGOL-like Compiler for SCHEME* S.B. Thesis, Massachusetts Institute of Technology, June 1984.

[80] **Guillermo Rozas** *Translucent Procedures, Abstraction without Opacity* Ph.D. Dissertation, Massachusetts Institute of Technology, October 1993. Also appears as MIT Technical Report AI-TR-1427.

[81] **Erik Ruf** *Partitioning Dataflow Analyses Using Types* In *ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, Paris, France, January 1997.

[82] **Brian C. Smith** *Reflection and Semantics in LISP* In *Proceedings of the 11th Annual ACM Symposium on Principles of Programming Languages*, Pages 23-35, December 1984.

[83] **Peter Solovitz** *Compilation for Fast Calculation over Pedigrees* In *Proceedings of Genetic Analysis Workshop 7, Cytogenetics and Cell Genetics*, Pages 136-138, S. Karger Medical and Scientific Publishers, 1992.

[84] **Raymond P. Stata** *Modularity in the Presence of Subclassing* Ph.D. Dissertation, Laboratory for Computer Science, MIT, May 1996.

[85] **Guy L. Steele Jr.** *Common LISP: The Language* Second Edition, Digital Press, 1990.

[86] **L. Sterling and E. Shapiro** *The Art of Prolog* MIT Press, 1981.

[87] **Bjarne Stroustrup** *The C++ Programming Language* Second Edition, Addison-Wesley, 1991.

[88] **Walter F. Tichy** *RCS - A System for Version Control* In *Software In Software - Practice and Experience*, 15(7), Pages 637-654, July 1985.

[89] **Elizabeth K. Turrisi** *Chapter and Verse Program Description* MIT AI Lab Working Paper 256, June 1984.

[90] **Amin M. Vahdat** *The Design of a Metaobject Protocol Controlling Behavior of a Scheme Interpreter* Xerox PARC Technical Report, January, 1993.

[91] **Jan L. A. Van de Snepscheut** *Proxac: An Editor for Program Transformation* Technical Report Caltech-CS-TR-93-33, California Institute of Technology 1993.

[92] **Rocke Verser** *DES Challenge* http://www.frii.com/~rcv/deschall.htm

[93] **Richard C. Waters** *The Programmer's Apprentice: A Session with KBEmacs* In *IEEE Transactions on Software Engineering* 11(11), Pages 1296-1320, November 1981.

[94] **Mark Weiser** *Program Slicing* In *IEEE Transactions on Software Engineering* 10(4), Pages 352-357, July 1984.

[95] **Deborah Whitfield and Mary Lou Soffa** *Automatic Generation of Global Optimizers* In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation* Toronto, Ontario, CA, June 1991.

[96] **Akinori Yonezawa and Brian C. Smith**, eds. *Proceedings of the International Workshop on New Models for Software Architectures: Reflection and Meta-Level Architecture* Tokyo, Japan, November 1992.