N^o d'ordre: 2379

THÈSE

Présentée devant

devant l'Université de Rennes 1

pour obtenir

le grade de : Docteur de l'Université de Rennes 1
Mention Informatique

par

Ulrik Schultz

Équipe d'accueil : IRISA
École Doctorale : Sciences Pour l'Ingénieur
Composante universitaire : IFSIC

Titre de la thèse :

Object-Oriented Software Engineering
using
Partial Evaluation

soutenue le 15 Decembre 2000 devant la commission d'examen

M. :	Jean-Pierre	Banâtre	Président
MM. :	Craig	Chambers	Rapporteurs
	Pierre	Cointe	
MM. :	Julia	Lawall	Examinateurs
	Gilles	Muller	
	Charles	Consel	

Abstract

Object-oriented programming languages facilitate mapping an abstract characterization of a problem into a generic, extensible implementation built from reusable program components. However, a well-designed object-oriented program may be overly general when used for a specific purpose, and hence be inefficient in terms of speed and size. To enhance performance and reduce program size, the program can be adapted to encompass only the specific functionality required in a specific situation.

Partial evaluation is an automatic program specialization technique, that adapts a program to a given execution context. In the object-oriented software development process, a program is developed by mapping a problem characterization into a generic implementation. Partial evaluation provides an additional stage in the object-oriented software development process: it maps a general implementation into a dedicated implementation. It is our thesis that partial evaluation can be integrated into the object-oriented software development process, as a software engineering tool that configures a generic program to function in a specific context; partial evaluation eliminates inefficiencies that otherwise would have been adressed by manually rewriting the program.

The contributions of this work are as follows:

- We identify typical overheads in generic object-oriented programs that can be eliminated using partial evaluation.

- We demonstrate how partial evaluation maps generic software patterns into dedicated, optimized implementations

- We improve the predictability of partial evaluation by the means of a tight coupling between partial evaluation and the software development process.

- We formally define partial evaluation for object-oriented languages.

- We design and develop a complete partial evaluator for Java.

- We provide extensive experiments that illustrate the performance advantage provided by partial evaluation.

Preface

This dissertation documents the research done during my PhD studies between October 1997 and October 2000 at IRISA, University of Rennes I, Rennes, Brittany, France. The first chapter gives an overview of the research reported in this document.

Acknowledgements

The members of the Compose group have been an invaluable aid throughout my entire thesis work. I am very grateful to Charles Consel, my thesis advisor, for his vigilant guidance of my path from student to researcher. Through numerous discussions and fruitful collaboration, the other members of the Compose group have helped this work to mature. I in particular wish to thank Julia Lawall, Renaud Marlet and Gilles Muller for their help. Furthermore, the implementation work done as part of my thesis is based directly on earlier work by other members of the Compose group, and could not have been completed without the assistance of certain members of the Compose group, namely Peter Chang, Neetu Nangia, and Miguel de Miguel.

The ideas that I develop in this thesis are rooted in existing work in the research community. Craig Chambers and Gregor Kiczales have in particular influenced my work by providing welcome input on critical points. My work has also benefited from discussions with Peter Bertelsen, who is conducting a similar study for his thesis work, and from my work with Helle Markmann Andersen, who has kindly let me supervise her DEA project in Rennes.

I would like to thank Jean-Pierre Banâtre, Craig Chambers, Pierre Cointe, Julia Lawall, and Gilles Muller for having accepted to participate in my jury.

The first three chapters of this document were translated to French by Mickaël Chaignault, and subsequently corrected with the help of Charles Consel, Anne-Françoise Le Meur, Fabrice Mérillon, Gilles Muller, and Laurent Réveillère.

I wish to dedicate this dissertation to my wife, Mette. Our three years in Rennes spent together in our small apartment, with her studies in philosophy, my research in computer science, and our favorite restaurants in the street below, will forever remain a treasured memory. My thanks also to my family, for their moral and material support.

This work is financed by Bull.

Contents

I Introduction 1

1 Introduction 3
- 1.1 Specialization of Object-Oriented Programs . . . 3
- 1.2 Example: Image Processing . . . 5
  - 1.2.1 Image filtering . . . 5
  - 1.2.2 Structure of the Implementation . . . 6
  - 1.2.3 Opportunities for Specialization . . . 7
- 1.3 Overview . . . 8

2 Object-Oriented Languages 11
- 2.1 Introduction . . . 11
- 2.2 Object-Oriented Features . . . 12
- 2.3 Class-Based Languages . . . 13
- 2.4 Object-Based Languages . . . 14
- 2.5 A Small Object-Oriented Language . . . 14
- 2.6 Summary . . . 16

3 Partial Evaluation 17
- 3.1 Introduction . . . 17
- 3.2 Principles of Partial Evaluation . . . 18
  - 3.2.1 Partial evaluation of a program . . . 18
  - 3.2.2 Partial evaluation for functional languages . . . 19
  - 3.2.3 Partial evaluation for imperative languages . . . 22
- 3.3 Specialization for Object-Oriented Languages . . . 23
  - 3.3.1 Inheritance . . . 24
  - 3.3.2 Class hierarchy specialization . . . 25
  - 3.3.3 Customization and selective argument specialization . . . 25
  - 3.3.4 Specialization classes . . . 26
- 3.4 Summary . . . 27

II Software Engineering and Partial Evaluation 29

4 Software Engineering 31
- 4.1 Introduction . . . 31
- 4.2 Software Engineering using Partial Evaluation . . . 32
  - 4.2.1 Scientific computations and computer graphics . . . 32

4.2.2 Operating systems 32
4.2.3 Software architectures 33
4.3 Assessment of Partial Evaluation 33
4.3.1 Partial evaluation pitfalls 33
4.3.2 Partial evaluation and software development 34
4.4 Object-Oriented Software Development 34
4.4.1 Developing object-oriented programs 35
4.4.2 Common overheads in object-oriented programs 35
4.4.3 Using partial evaluation 35
4.5 Design Patterns 36
4.5.1 Capturing object-oriented designs 36
4.5.2 Overheads from using design patterns 37
4.5.3 Using partial evaluation 38
4.6 Summary 38

5 Object-Oriented Partial Evaluation 41
5.1 Introduction 41
5.2 Object-Oriented Partial Evaluation 42
5.2.1 Specializing an object-oriented program 42
5.2.2 Specializing object-oriented mechanisms 44
5.2.3 Speculative specialization 45
5.2.4 Appearance of the specialized program 45
5.2.5 Example: power 46
5.3 Aspects and Specialized Program Appearance 47
5.3.1 Modularity and partial evaluation 47
5.3.2 Aspect-oriented programming and partial evaluation . . 49
5.3.3 Specialization as an aspect 50
5.4 Specialization Examples 52
5.4.1 Power revisited 52
5.4.2 Arithmetic expression interpreter 52
5.5 Summary 53

6 Declarative Specialization 57
6.1 Introduction 57
6.2 Specialization Classes 58
6.2.1 Specialization class basics 59
6.2.2 Combining specialization classes 59
6.2.3 Specialization class example 60
6.3 Extended Specialization Classes 60
6.3.1 Extensions 61
6.3.2 Syntax and semantics 61
6.3.3 Examples 65
6.3.4 Implementation 66
6.4 Specialization Classes as Aspects 67
6.5 Software Development and Specialization Classes 68
6.6 Summary 68

7 Specialization Patterns **71**
7.1 Introduction . . . 71
7.2 Partial Evaluation Pitfalls . . . 72
7.2.1 Directing partial evaluation . . . 72
7.2.2 Limitations of partial evaluation . . . 72
7.3 Specialization Patterns . . . 73
7.3.1 Specialization patterns: definition and use . . . 73
7.3.2 Creational patterns . . . 75
7.3.3 Structural patterns . . . 76
7.3.4 Behavioral patterns . . . 79
7.4 A Complete Example . . . 82
7.4.1 Description . . . 83
7.4.2 Overheads . . . 83
7.4.3 Specialization . . . 84
7.5 Assessment . . . 84
7.5.1 Software engineering and specialization patterns . . . 85
7.5.2 Performance issues . . . 85
7.6 Summary . . . 86

III Object-Oriented Partial Evaluation **87**

8 Formalization of Object-Oriented Partial Evaluation **89**
8.1 Introduction . . . 89
8.2 Language: Extended Featherweight Java . . . 90
8.2.1 Language concerns . . . 90
8.2.2 EFJ syntax . . . 90
8.2.3 EFJ typing . . . 92
8.2.4 EFJ reduction . . . 94
8.3 Two-level Language . . . 95
8.4 Well-Annotatedness . . . 96
8.4.1 Binding times . . . 97
8.4.2 Considerations . . . 97
8.4.3 Well-annotatedness rules . . . 97
8.5 Specialization . . . 100
8.5.1 2EFJ expression reduction . . . 100
8.5.2 Reduction of static expressions . . . 100
8.5.3 Reduction of dynamic expressions . . . 102
8.5.4 Reduction of a program . . . 102
8.5.5 Improved 2EFJ expression reduction . . . 103
8.6 Binding-Time Analysis . . . 104
8.6.1 Constraint system . . . 104
8.6.2 Constraint generation . . . 105
8.6.3 Constraint solving . . . 106
8.7 Examples . . . 107
8.7.1 The power example . . . 107
8.7.2 The fold example . . . 108

8.8 Dealing with Object-Based Languages . . . 111
8.8.1 Considerations . . . 112
8.8.2 Specialization . . . 113
8.8.3 Binding-time analysis . . . 113
8.9 Summary . . . 114

9 Partial Evaluation of Realistic Object-Oriented Programs 115
9.1 Introduction . . . 115
9.2 Use-Sensitivity . . . 117
9.2.1 Use-sensitive binding times . . . 118
9.2.2 Use-sensitivity for object-oriented languages . . . 119
9.2.3 Use-sensitive binding-time analysis . . . 119
9.2.4 Use-sensitive specialization . . . 120
9.3 Polyvariant Analysis . . . 121
9.3.1 Type-polyvariant binding times . . . 121
9.3.2 Method-polyvariant binding times . . . 122
9.3.3 Polyvariant alias information . . . 123
9.3.4 Controlling polyvariance . . . 124
9.4 Instantiation-Time Specialization . . . 124
9.4.1 Instantiation-time specialization: what . . . 124
9.4.2 Instantiation-time specialization: how . . . 126
9.4.3 ITS implementation . . . 126
9.4.4 Instantiation-time specialization safety . . . 127
9.4.5 Perspectives on instantiation-time specialization . . . 128
9.5 Modular Specialization . . . 129
9.5.1 Program slicing granularity . . . 129
9.5.2 Behavior of external methods . . . 130
9.6 Post-Specialization Optimizations . . . 131
9.6.1 Post-specialization optimization . . . 131
9.6.2 Optimizations for object-oriented languages . . . 131
9.7 Summary . . . 133

10 Partial Evaluation for Java 135
10.1 Introduction . . . 135
10.2 Overview of the Java Language . . . 136
10.3 JSpec: A Partial Evaluator for Java . . . 137
10.3.1 Features . . . 137
10.3.2 Specializing object-oriented language features . . . 139
10.3.3 Specializing non-object language features . . . 142
10.4 JSpec Implementation . . . 143
10.4.1 Architecture . . . 143
10.4.2 Java-to-C translation . . . 145
10.4.3 C partial evaluation . . . 149
10.4.4 C-to-Java translation . . . 152
10.4.5 Shortcomings of the current implementation . . . 154
10.5 Software Engineering Experiences . . . 156
10.5.1 C as an intermediate language . . . 156

10.5.2 Proposal for a revised design 157
10.6 Summary 158

11 Experimental Study 159
11.1 Introduction 159
11.2 Compiler Optimizations vs. Partial Evaluation 160
11.2.1 Object-oriented compiler optimizations 160
11.2.2 Specializing for program speed 161
11.3 Java Execution Models 161
11.4 Description of Benchmark Programs 163
11.4.1 Imperative opportunities 163
11.4.2 Object-oriented opportunities 164
11.4.3 Mixed opportunities 165
11.5 Experiments 165
11.5.1 Methodology 166
11.5.2 Java-to-Java specialization results 166
11.5.3 Java-to-C specialization results 174
11.6 Assessment 174
11.6.1 Impact of the kind of specialization opportunity 174
11.6.2 Impact of execution model and architecture 175
11.7 Summary 177

IV Perspectives 178

12 Related Work 180
12.1 Introduction 180
12.2 Object-Oriented Partial Evaluation 180
12.2.1 Partial evaluation 181
12.2.2 Application: fault-tolerant computing 182
12.3 Partial Evaluation Principles 182
12.3.1 Specialization via translation 183
12.3.2 Specialization of reflection 183
12.3.3 Optimizing method dispatching 183
12.4 Optimizing Compilers 183
12.4.1 Selective argument specialization 184
12.4.2 Rewrite-system based compilation 185
12.4.3 Partial evaluation in a compiler 185
12.5 C++ Templates 186
12.6 Specialization Patterns 186
12.6.1 Software architectures 187
12.6.2 Specialization recipes 187
12.7 Summary 187

13 Future Work **188**
13.1 Introduction 188
13.2 Concepts in Object-Oriented Partial Evaluation 188
13.2.1 Correctness issues 188
13.2.2 Language features 189
13.2.3 Partial evaluation features 189
13.2.4 Program size optimization 190
13.3 Continued JSpec Development 190
13.4 Partial Evaluation and Software Engineering 191
13.4.1 Tool support for partial evaluation 191
13.4.2 Specialization patterns 192
13.4.3 Partial evaluation for software components 192
13.5 Summary 192

14 Conclusion **194**
14.1 Object-Oriented Partial Evaluation 194
14.2 Partial Evaluation for Java 195
14.3 Software Engineering 195
14.4 Final Remarks 195

A Implementation and Formalization Details **196**
A.1 Example Implementation Details 196
A.2 Formalization Details 196

List of Figures

1.1 Core of the image processing program. 6
1.2 Specialized image processing program. 7
1.3 Chapter dependency graph. 9

2.1 Hierarchy of binary operators. 15
2.2 Generic power function. 16

3.1 Generic power program in Scheme. 21
3.2 Specialized versions of power program of Figure 3.1. 22
3.3 Generic power program in C. 23
3.4 Specialized versions of power program of Figure 3.3. 24

4.1 A standard use of the iterator design pattern. 37

5.1 The power example (again). 42
5.2 Generic interaction with `Power` object. 43
5.3 Specialization invariant and resulting interaction. 43
5.4 Specialization of the power example. 46
5.5 Specialization producing unsafe methods. 48
5.6 Syntax of an aspect language for expressing specialized programs. 51
5.7 Specialized code of Figure 5.5 as an aspect. 51
5.8 Specialized power in different versions. 53
5.9 Arithmetic expression interpreter. 54
5.10 Specialization of arithmetic expression interpreter. 54

6.1 Specialization class for power example. 60
6.2 Grammar of extended specialization classes. 62
6.3 Specialization class for power example (revised). 65
6.4 Specialization classes for arithmetic expression interpreter example. 66

7.1 Specialization pattern template. 74
7.2 Overview of specialization process. 74
7.3 Specializing a use of the builder pattern. 76
7.4 Relevant parts of `AbstractList` and the implementation `LList`. . 77
7.5 Builder specialization pattern. 78
7.6 Specializing a use of the bridge pattern. 79
7.7 Bridge specialization pattern. 80
7.8 Specializing a use of the strategy pattern. 81

7.9 Strategy specialization pattern. 82
7.10 Overview of graphical application. 83
7.11 Call graph of `repaint()`, before and after specialization. 84

8.1 EFJ syntax (program and values) 91
8.2 EFJ auxiliary definitions 92
8.3 EFJ subtyping 92
8.4 EFJ typing 93
8.5 EFJ computation 94
8.6 2EFJ syntax 95
8.7 Rules for well-annotatedness 98
8.8 Specialization as two-level execution 101
8.9 Auxiliary definitions for Figure 8.8 102
8.10 Specialization of a program into an aspect 103
8.11 Constraint generation 105
8.12 The power example (again) 107
8.13 Binding-time annotated version of the power example 108
8.14 Folding over lists 109
8.15 Binding-time annotated version of the fold example with static list 110
8.16 Specialization for list {`6,4`} 110
8.17 Binding-time annotated version of the fold example with static operator and neutral value 111
8.18 Specialization for operator `Add` and initial value `0` 112

9.1 Use-sensitivity of binding times. 117
9.2 Statically constructed object used under dynamic control. 125
9.3 Instantiation-time specialized version of Figure 9.2. 127

10.1 Speculative specialization of interface call. 140
10.2 Specialization of constructors. 141
10.3 JSpec architecture. 144
10.4 Translation of a Java class. 148
10.5 Translation of selected Java statements. 149
10.6 Partial evaluation of translated statements. 151
10.7 Back-translation of specialized statements into Java. 153

11.1 Speedup comparison, SPARC. 171
11.2 Speedup comparison, x86. 173

A.1 Relevant parts of `Array` and its iterator `ArrayIterator`. 197
A.2 Functional language self-interpreter in Scheme. 198
A.3 Auxiliary definitions for Figure 8.7. 199
A.4 Specialization as two-level execution, with a method cache. . . . 200
A.5 Auxiliary definitions for Figure A.4. 201
A.6 Delta reduction rules for EFJ. 201

List of Tables

11.1 Specialization effect on code size measured in number of lines. . . 168
11.2 Specialization benchmarks, interpreter execution. 168
11.3 Specialization benchmarks, JIT execution. 169
11.4 Specialization benchmarks, adaptive compiler execution. 170
11.5 Specialization benchmarks, off-line compiler execution. 172
11.6 Specialization with C source code as output (execution under Harissa). 172

Part I

Introduction

Chapter 1

Introduction

1.1 Specialization of Object-Oriented Programs

One of the major advantages of object-oriented languages is the ease with which real-world problems can be modeled and implemented in terms of objects. The direct correspondence between the problem domain and the program facilitates translation of the problem into an initial implementation design. Conceptually, an object-oriented program is implemented in terms of autonomous interacting objects. The more generic and parameterized an object, both in terms of internal parameters and in terms of usage of other objects, the more it can be directly reused in different programs. Reusable program components are created easily in object-oriented languages, which has been a major factor in their wide adoption as general-purpose programming languages.

A reusable program component will often contain features not needed in a specific situation. Indeed, many programs do not fully exploit the genericity offered by a reusable object. Certain parts of an object, or parts of the context in which an object is used, may be fixed when the object is created, or may simply remain fixed for a period of time. When this is the case, any decisions and computations in the program that are controlled by this fixed information are themselves fixed; they may be performed over and over again, which translates into a loss of performance. This conflict between software generality and performance has been recognized in areas such as operating systems [35] and graphics [100].

The generality versus performance conflict is being increasingly addressed with success, using forms of *program specialization* [106]. Program specialization is the adaptation of a generic program component to a given usage context, based on arbitrary invariants about the usage context. This approach can lead to considerable performance gains, by eliminating from the general code all aspects not related to a given context. Program specialization has been manually performed by adapting critical program components to the most common usage patterns [15, 134, 135]. Manual specialization improves performance, but has a limited applicability, because the process is error prone.

Automatic tools have been developed to specialize programs to a given context [6, 11, 12, 40, 41, 93, 98], using a technique referred to as *partial evaluation.*

Partial evaluation is an automatic technique for specializing a program to its execution context [84]. Partial evaluation adapts a generic program to information about its usage, in the form of known input parameters or parameters implicitly fixed in the program. It propagates known information about the usage context throughout the program, and any computation that depends solely on known information is reduced away. All that remains after specialization are the computations that depend on unknown information. Partial evaluation has been investigated thoroughly for functional [21, 40], logical [99] and imperative [6, 12, 41] languages, and has recently been discussed for object-oriented languages by Schultz et al., in the context of a prototype partial evaluator for Java [143]. Applications of partial evaluation have emerged in a number of fields, including scientific code [12, 14], systems software [41, 59, 116], and computer graphics [70], with very promising results.

We apply partial evaluation within the object-oriented paradigm to reconcile the opposing needs of genericity and efficiency. This document provides a complete presentation of partial evaluation for object-oriented languages. As is the case for functional, logical, and imperative languages, partial evaluation can specialize a program written in an object-oriented language. Partial evaluation complements the object-oriented software engineering process: object-oriented languages facilitate mapping a problem analysis into a generic implementation, and partial evaluation maps such a generic object-oriented program into an efficient implementation.

It is our thesis that partial evaluation can be integrated into the object-oriented software development process, as a software engineering tool that configures a generic program to function in a specific context; partial evaluation eliminates inefficiencies that otherwise would have been addressed by manually rewriting the program. Our contributions are as follows:

- We identify generic mechanisms in object-oriented programs that constitute recurring opportunities for specialization.
- We provide a conceptual account of how partial evaluation exploits these specialization opportunities by specializing object-oriented programs.
- We formally define an off-line partial evaluator for a minimal object-oriented language, including a binding-time analysis and a specializer.
- We improve the predictability of partial evaluation by giving guidelines for using partial evaluation as a software engineering tool to map a generic design into an efficient implementation.
- We define a conceptual approach for linking experience with using partial evaluation to specific program constructions.
- We have designed and developed a complete partial evaluator for Java, and we demonstrate how to partially evaluate Java programs.
- We describe those features of a partial evaluator that we have found to be essential when treating realistic object-oriented programs.

- We measure the advantages of partial evaluation through an extensive experimental study of the performance of programs specialized using our partial evaluator across various execution platforms.
- We extend earlier results on controlling partial evaluation for object-oriented languages using a declarative framework [162], by extending this framework to handle examples written in an object-oriented style of programming.

To illustrate how partial evaluation works and how it transforms an object-oriented program, Section 1.2 presents an example of an object-oriented application architectured using generic object-oriented abstractions; this application is specialized into an efficient implementation using partial evaluation. Afterwards, Section 1.3 provides an overview of the entire document, and discusses prerequisites and terminology.

1.2 Example: Image Processing

An object-oriented program can be structured to enhance genericity by composing it from individual objects that interact through generic interfaces. As an example of such a program, we present an image filtering program. We have chosen an ideal design in terms of genericity; with a single program we model a family of related image-processing problems. Partial evaluation is used to adapt this generic program to parameters specific to a given image-processing problem, which results in an efficient specialized program dedicated to a given task.

We use the image processing example to illustrate two important points. First, specialization opportunities in object-oriented programs may depend heavily on object-oriented mechanisms. Second, when a generic object-oriented program is appropriately structured, partial evaluation can map the program into an efficient implementation. The example exploits well-known object-oriented designs such as the strategy and abstract factory design patterns [64]. The use of these designs facilitates program development but impedes compiler optimizations. This section first introduces image filtering, then describes the structure of the example illustrating the generic implementation, and finally discusses the opportunities that we find for specialization.

1.2.1 Image filtering

We consider image filtering based on a matrix, known as a *mask*. Filtering is performed by moving the mask across an image, computing the filtered pixel in the position of the center of the mask by performing operations on the pixels covered by the mask. Such filters can be used to obtain a variety of effects, including blurring, edge detection, and noise elimination [141].

There is a variety of possible representations for the data that defines an image; each representation has specific advantages and disadvantages. For this reason, all image data and specific pixels extracted from an image are manipulated as abstract data by the filtering process; this design makes it possible to

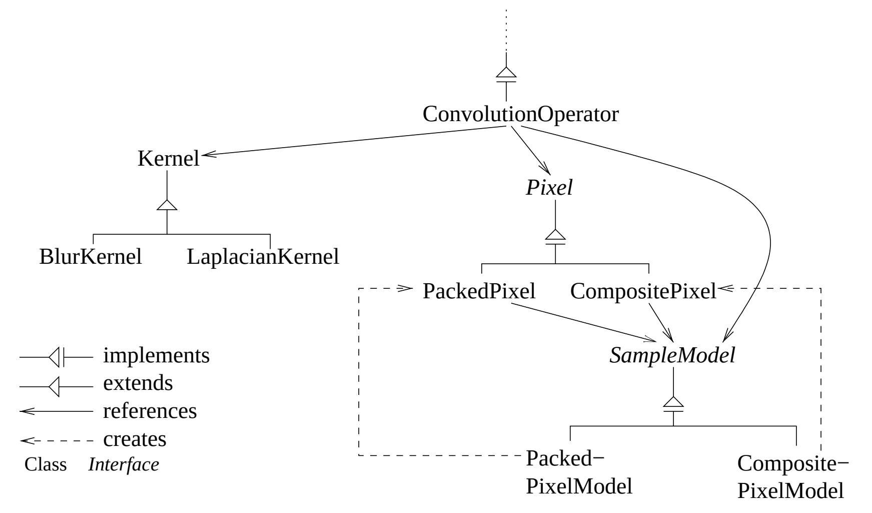


Figure 1.1: Core of the image processing program.

choose the most efficient data representation for an image independently of the choice of the image filter.

1.2.2 Structure of the Implementation

Based on our considerations about program genericity and image processing, we have chosen a design where multiple layers of abstraction facilitate extensibility and maintenance of a wide range of functionalities. An image is viewed as an abstract source of pixels, and sources are filtered by abstract operators that are applied to each pixel. Filtered images are themselves pixel sources. Pixel operations are decoupled from their low-level representation by an abstract interface.

The structure of the core of the example program is shown with an object diagram in Figure 1.1. The program delegates data manipulation to a pixel processing strategy defined as an image operator, which among other classes is implemented by the `ConvolutionOperator` class. A `ConvolutionOperator` object computes a linear combination of the pixels covered by its mask; the mask is defined by a `Kernel` object. The behavior of a `ConvolutionOperator` object is determined by the data contained in the `Kernel` object: the nature of the transformation performed by the operator can be changed by modifying or replacing the `Kernel` object.

The image data processed by an operator is accessed through `Pixel` objects, which are instantiated to match the data representation of the image currently being transformed. A concrete `Pixel` object is instantiated using `SampleModel` as an abstract factory. A `SampleModel` object is associated with each pixel source. The `PackedPixelModel` class is an implementation of `SampleModel` where multiple

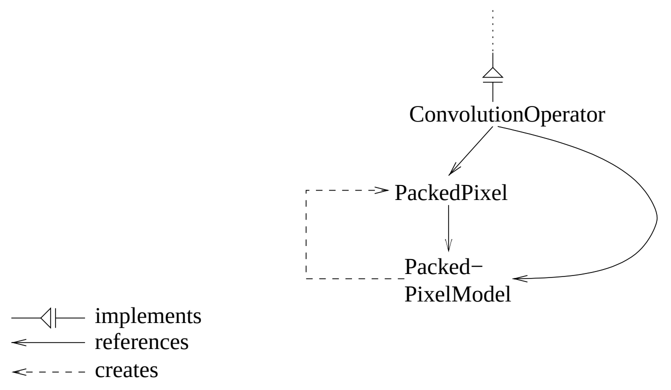


Figure 1.2: Specialized image processing program.

samples (color components, for example) are packed into a single integer value and where individual pixels are manipulated through `PackedPixel` objects. In `CompositePixelModel` each kind of sample is stored separately, and pixels are manipulated through `CompositePixel` objects. In this way, all image operations performed by a `ConvolutionOperator` object are done in terms of the abstract `Pixel` interface, and the `SampleModel` object associated with the pixel source controls all aspects of image data access.

1.2.3 Opportunities for Specialization

The parameterization of the image processing algorithm and its separation into distinct classes offers many advantages in terms of ease of implementation and future extensibility. However, it induces a heavy performance penalty; even when limited to a small blurring mask, this implementation will perform 50 virtual calls to filter a single pixel! A hand-optimized implementation, where a dedicated filter is programmed independently for each kind of operator and data representation, would perform no virtual calls. Indeed, image processing applications are rarely structured with as much genericity as we have chosen for this application. Instead, efficiency is enhanced at the price of genericity and ease of maintenance. As an alternative, partial evaluation can be applied to the program to enhance performance.

The image filtering program is structured to allow flexibility in specifying the filter to be applied to the image, and the concrete representation of the pixels of the image. Nevertheless, once we begin applying a particular filter to a particular image, both the filter and the pixel representation remain invariant. As a result, the decision of what filter and pixel representation to use is repeated needlessly throughout the filtering of an image, which is a significant opportunity for specialization. First, we can specialize the program to a specific kernel, by fixing the behavior of the `ConvolutionOperator` object. This can be done by adapting it to a concrete `Kernel` object. Second, we can specialize the program to a specific representation, which allows the concrete `Pixel` representation and `SampleModel` object to be accessed directly. The object diagram of the resulting program structure is shown in Figure 1.2. The kernel object was no longer needed after specialization, and has been eliminated. The `ConvolutionOperator` object refers directly to the remaining objects, facilitating subsequent compiler optimizations. The specialized program is from 1.5 to 5 times faster than the original [143] (all experiments are described in Chapter 11).

1.3 Overview

This section presents an overview of the remaining document, lists the prerequisites for reading this document, and discusses various issues pertaining to terminology.

Chapter overview

The remainder of the introductory part (Part I) presents background material. We first describe the primary features of object-oriented languages (Chapter 2). Then we explain the basic concepts of partial evaluation, and give an overview of existing techniques for specialization of object-oriented programs (Chapter 3).

We are interested in partial evaluation as a tool for object-oriented software engineering. We initially concentrate on conceptual issues in partial evaluation as a software engineering tool (Part II). First, we assess the utility of partial evaluation as a software engineering tool, and discuss partial evaluation in the context of object-oriented software engineering (Chapter 4). Then, we define the conceptual nature of partial evaluation for object-oriented languages (Chapter 5). With the basic object-oriented partial evaluation concepts defined, we explore a declarative language for controlling the application of partial evaluation to object-oriented programs (Chapter 6), and describe how knowledge about specific specialization scenarios can be communicated within the partial evaluation community (Chapter 7).

For a complete understanding of partial evaluation for object-oriented languages, an in-depth treatment of various technical issues is needed. To this end, we provide a detailed presentation of principles and practices in partial evaluation for object-oriented languages (Part III). First, we formalize partial evaluation for object-oriented languages (Chapter 8). Then, we discuss what features are required from a partial evaluator to handle realistic object-oriented programs (Chapter 9), and present our implementation of a complete partial evaluator for Java (Chapter 10). Last, we explore partial evaluation in its traditional role, as a means of optimizing programs; for completeness, we compare results across various execution environments (Chapter 11).

To conclude the document, we present perspectives on our work (Part IV). We present related work (Chapter 12), future work (Chapter 13), and finally our conclusions (Chapter 14).

Some of the more complex concepts and techniques presented in this document are introduced gradually over several chapters or are based on information

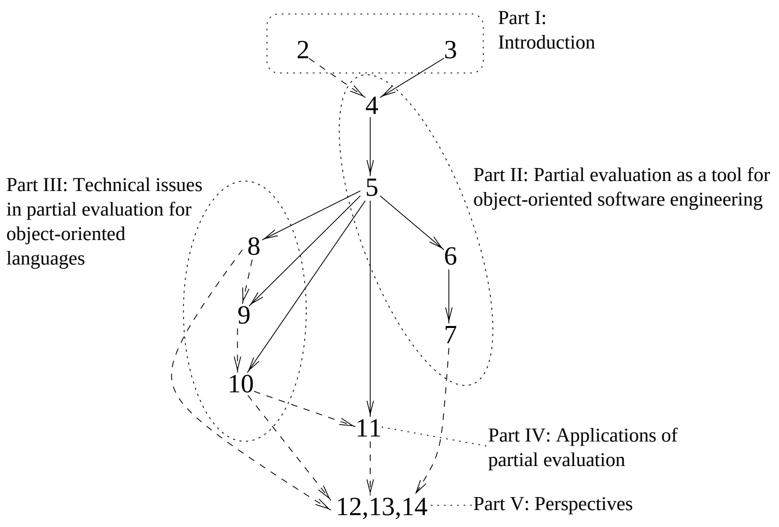

Part I:
Introduction
2
3
4
5
Part II: Partial evaluation as a tool for object-oriented software engineering
Part III: Technical issues in partial evaluation for object-oriented languages
8
9
10
6
7
11
Part IV: Applications of partial evaluation
12,13,14
Part V: Perspectives

Figure 1.3: Chapter dependency graph.

in an earlier chapter. The reader not wishing to study every aspect of this document may prefer to skip certain chapters. To ensure an easy reading, the Chapter dependencies shown in Figure 1.3 should be respected (a dotted arrow indicates a partial dependency that improves understanding but can be ignored).

Prerequisites

Although mostly self-contained, there are certain prerequisites for a complete understanding of this document. Chapters 2 and 3 provide sufficient background material for an understanding of the conceptual presentation of partial evaluation for object-oriented languages in Chapter 5. However, basic knowledge of formal semantics and familiarity with formalizations of partial evaluation for functional or imperative languages is required for a complete understanding of the formalization of partial evaluation for object-oriented languages in Chapter 8. Likewise, a basic knowledge of the features of the Java language is required for a complete understanding of the Java partial evaluator presented in Chapter 10.

Terminology

Having mostly evolved separately, the terminology used in the fields of object-oriented programming and partial evaluation overlap in an incompatible fashion. Specifically, there are two words, “static” and “specialize” that must be disambiguated. In both cases there are alternative words available in object-oriented programming.

- In some object-oriented languages, class fields and class methods (fields and classes defined in a class without a relation to a specific object instance of the class) are referred to as static fields and static methods. However, the word "static" is used in partial evaluation to indicate known information. To resolve the conflict, we refer to such fields and methods as class fields and class methods. The word static will only be used in conjunction with object parts to describe known information (i.e., a static field is always a field that contains known information).

- A subclass is often said to "specialize" its superclass. There are however many forms of specialization that can be applied to a program. To avoid confusion, we refer to the relation between a subclass and its superclass in terms of inheritance (the subclass inherits from the superclass) or in terms of the subclass/superclass relation (one class subclasses some other class, or one class is the superclass of some other class). The word "specialization" is often used in the context of partial evaluation, and here means specialization as defined by partial evaluation.

Apart from these two exceptions, standard object-oriented vocabulary will be used throughout this document.

The terms "method unfolding" and "method inlining" can be considered synonyms; in both cases is the definition of a method inserted into its calling context. However, we use "method unfolding" to mean a transformation performed during partial evaluation, and "method inlining" to mean a compiler optimization that may be performed if it is considered an advantage.

Chapter 2

Object-Oriented Languages

2.1 Introduction

In object-oriented programming, a program execution is viewed as a computerized physical model that simulates the behavior of either a real or imaginary part of the world [102]. In a computerized physical model, objects are the material used for representing or modeling physical phenomena from the application domain. Objects are characterized by various attributes, that represent or model a property of the phenomenon being modeled. The state of an object represents the variation at different points in time of those attributes that are measurable, and the actions associated with an object can change the state of the object and interact with other objects. As is the case for real-world entities, objects can be organized by classifying them according to their properties, which naturally leads to classification hierarchies. Similarly, objects can be understood in terms of their decomposition into part objects, or in terms of their relations to other objects.

In practice, object-oriented languages offer numerous advantages. First, the strong analogy between the software model and the physical model offers a systematic and thoroughly researched way of developing software to solve a given problem. Second, objects form natural data abstraction barriers; the isolation of individual program features in behaviorally autonomous objects improves design resilience when changes are made to some part of the system during its evolution. Last, the natural organization of objects into hierarchies and compositions facilitates the isolation of common functionality in a single place, thus improving code reusability. These advantages have caused a wide adoption of object-oriented languages throughout academia and industry.

Object-oriented languages can roughly be divided into two categories: class-based languages and object-based languages. In both kinds of languages, objects encapsulate state and actions, and object composition forms relations between objects. Class-based languages are centered around the notion of classes as descriptions of objects. Classes organize objects into hierarchies according to common purpose and attributes. Object-based languages are conceptually simpler in that object composition is the sole means of organizing relations between objects; objects fulfill the purpose of classes when appropriate.

The purpose of this chapter is to provide an overview of object-oriented languages, and describe those features of object-oriented languages that we consider important. It is essential to have a complete understanding of the basic features of object oriented languages before we proceed to discuss and define partial evaluation for object-oriented languages. To ensure an easy understanding of the program examples used in this document, this chapter also provides an informal overview of a small object-oriented language based on Java, which will be used later to formalize partial evaluation for object-oriented languages.

Chapter overview: First, Section 2.2 describes features common to object-oriented languages. Then, Section 2.3 provides an overview of class-based object-oriented languages, and Section 2.4 provides an overview of object-based languages. Last, Section 2.5 describes a small object-oriented language to be used later for examples, and Section 2.6 summarizes the contents of this chapter.

2.2 Object-Oriented Features

Object-oriented languages share a number of programming-language features and programming principles that allow the programmer to express solutions in terms of individual, interacting objects.

An object encapsulates a number of attributes that define its state, as well as a number of attributes that define the actions of the object. We refer to state-defining attributes as *fields*, and action-defining attributes as *methods*. During the evaluation of a method, the object to which the method belongs is referred to as the *self*. The self is used to access fields of the object. It is the explicit syntactic and semantic association between fields and methods in an object that ensures the autonomous nature of objects. Some languages support the use of *access modifiers* to control access to object attributes (for example C++ [147], Eiffel [111], and Java [67]). Access modifiers are not fundamental to object-oriented languages. Nonetheless, they impose limitations on the interaction that may take place between objects, and so must be taken into account when defining program transformations that modify object interaction.

Objects interact by *message passing*, where a named message and a set of arguments are passed from one object (the sender) to another (the receiver). The method corresponding to the message name is invoked, which binds its formal parameters to the arguments. Message passing is also referred to as virtual dispatching or as performing a virtual call. With ordinary dispatching (i.e., procedure invocation in an imperative language), it is explicit what method is called at a given call site, whereas with virtual dispatching, there is a set of possible callee methods for a given call site.

In a statically typed object-oriented language, a message may be sent to an object only if its type indicates that it is capable of handling the message. The *qualifying type* of an object is the type through which an object is manipulated; the *concrete type* is the actual type of the object. These types are related by the notion of *subtyping*. Intuitively, a type T is a subtype of a type T' if an element of type T can safely be used in a context where an entity of type T' is expected. Hence, to ensure static typing, the concrete type should be a subtype of the qualifying type.

Objects can be composed together to form compound structures. An object can have other objects as fields, and can reference other independent objects. Some languages, such as Java, only support references for composing objects, and so we will not consider objects that contain other objects as an essential object-oriented feature (in any case, references can mostly simulate objects containing other objects). An object can use some other object that it references (is composed with) by directly manipulating its fields or passing it a message. Object composition permits the functionality of an object to be expressed in terms of functionality provided by other objects. In this case, the former object is said to *delegate* functionality to the latter objects. Delegation facilitates writing programs that evolve over time: by modifying the composition of an object structure, the overall functionality of the object structure can be adjusted or changed.

2.3 Class-Based Languages

Class-based languages are the most common object-oriented languages in use throughout research and industry; prominent examples include Beta [102], C++ [147], CLOS [19], Eiffel [111], Java [67], Simula [121], and Smalltalk [65]. Although there are significant differences between most class-based languages, they share a fundamental set of common features, which are described in this section.

In a class-based object-oriented language, classes model concepts in the problem domain, and are seen as descriptions of objects, the concrete phenomena in the problem domain. A new object instance is created from a class. The object is classified as being an instance of the class, and the fields and methods of the object are described by the class: the fields define the state specific to the object and are thus necessarily associated with the object instance, whereas the methods define class-wide behavior and thus are shared by all instances of the class.

Classes can be organized into an *inheritance hierarchy:* a class can inherit from some other class, making it a subclass of this other class, its superclass. Any given class consists of the attributes that it defines, in addition to any attributes defined in its superclass (and, transitively, in the superclass of the superclass). When viewed as a classification hierarchy, a subclass is more specific than its superclass, since it defines more attributes and thus is classified more precisely.

In many statically typed languages, objects instantiated from two classes ordered by inheritance are ordered likewise in terms of subtyping. An object instantiated from a more specific class contains at least the same set of attributes, and can thus be used in any context where an object instantiated from a more general class can be used. Indeed, in some object-oriented languages, there is a one-to-one correspondence between the class inheritance hierarchy and the subtyping hierarchy.

2.4 Object-Based Languages

Object-based languages are relatively few compared to the plethora of class-based languages, and they are to some extent still evolving. Nonetheless, object-based languages offer an interesting mix of simplicity and versatility, and sidestep many complications common to class-based languages. Prominent examples of object-based object-oriented languages include Cecil [30, 32], Emerald [137], and Self [157]. This section provides a description of the basic principles of object-based object-oriented languages.

In an object-based language, there are no classes, only objects. A new object can be created either by cloning an existing object, copying all of its attributes, or by creating a new attribute-less object. An object can be modified not only by changing the value of its fields, but also by changing what attributes (fields and methods) are contained in the object. When appropriate, an object can play the role of a class by acting as prototype used to create (through cloning) all instances of a given "class," but with the additional twist that the "class" (the prototype) can be modified. Code reuse between objects similar to inheritance can be modeled through delegation. In terms of typing, types exist independently of objects, and so object-based languages offer a complete separation between implementation and interface.

The relative simplicity of object-based languages makes it easy to define a minimal calculus describing computations in an object-based language. For example, the σ-calculus simply defines an object as a collection of updatable methods [1]. Evaluation is defined in terms of two basic reductions: method invocation and method update. Method invocation sends a named message of no arguments to an object; it is performed by substituting the receiver inside the body of the method. Method update adds a new method to an object, replacing any method of the same name. Fields are represented by methods returning constant values.

2.5 A Small Object-Oriented Language

We now describe a small class-based object-oriented language based on Java [67], that will be used to provide examples. In Chapter 8 we will define syntax, formal semantics, and typing rules for this language; for now, we informally describe the language using a small example. Our language is named Extended Featherweight Java (EFJ, after Featherweight Java [81]), and is a strict subset of Java in terms of syntax and semantics. It incorporates classes and inheritance in a statically typed setting, fields and methods with formal parameters, object constructors, and conditionals. There are no side-effects on fields nor formal parameters, and recursion is the only means of expressing repetition. EFJ is intended to constitute a least common denominator for class-based languages so that any partial evaluation principles developed for EFJ will apply to most other class-based languages as well.

As an example illustrating the EFJ syntax, Figure 2.1 shows a collection of three classes, `Binary`, `Add`, and `Mult`. The class `Binary` defines a standard

```
class Binary extends Object {
  Binary() { super(); }
  int eval( int x, int y ) {
    return this.eval( x, y );
  }
}
class Add extends Binary {
  Add() { super(); }
  int eval( int x, int y ) {
    return x+y;
  }
}
class Mult extends Binary {
  Mult() { super(); }
  int eval( int x, int y ) {
    return x*y;
  }
}
```

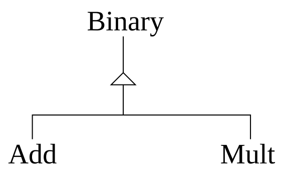


Figure 2.1: Hierarchy of binary operators.

interface for the concrete binary operators `Add` and `Mult`; its `eval` method of two integer arguments is defined to directly call itself and thus diverge. For the lack of abstract methods in the language, this is how we define an interface. The `Binary` class has a parameter-less constructor named after the class, which simply calls the superclass constructor. The `Add` and `Mult` classes both inherit from `Binary` (which permits them to be used in any context qualified by `Binary`), and define more interesting `eval` methods.

The `Power` class shown in Figure 2.2 can be used to apply a `Binary` operator a number of times to a base value. The constructor binds the exponent (the number of times to apply the operator), the operator, and the neutral value (the result of lifting a value to the 0^{th} power) to the appropriate fields. The syntactic form `x.f` is a reference to the field `f` of some object `x`, and the special variable `this` is bound to the self object. The `raise` method raises some base value to the exponent, using the auxiliary method `loop`; the syntactic form `x.m(a1,a2,...)` is an invocation of the virtual method `m` on the object `x` with arguments "`a1,a2,...`". The method `loop` uses the conditional form `b?x:y` (if `b` then `x` else `y`) to test the exponent value. The `loop` method returns the neutral value when the exponent is zero; otherwise it returns the result of applying the operator to the base and the result of a recursive call to `loop` with a decremented exponent. The `Power` class can be used as follows:

```
(new Power( y, new Mult(), 1 )).raise( x )
```

to compute x^y; the `new` operator creates a new instance of a class with fields initialized to the arguments passed to the constructor.

```
class Power extends Object {
  int exp; Binary op; int neutral;
  Power( int exp, Binary op, int neutral ) {
    super(); this.exp = exp; this.op = op; this.neutral = neutral;
  }
  int raise( int base ) {
    return loop( base, exp );
  }
  int loop( int base, int e ) {
    return e==0 ? this.neutral
                : this.op.eval( base, this.loop( base, e-1 ) );
  }
}
```

Figure 2.2: Generic power function.

2.6 Summary

Object-oriented languages propose major advantages in terms of software development, which has led to their wide adoption throughout academia and industry. With object-oriented languages, an application domain can be systematically mapped into a generic implementation. To a large extent, this implementation can be created from reusable components. Encapsulation enhances software resilience; message passing and delegation improves program flexibility; the natural support for classification hierarchies facilitates code organization. Class-based languages are centered around classes as the means of describing concepts that are instantiated into objects; object-based languages offer simplicity and a more dynamic programming model by replacing the special class construction by plain objects.

Chapter 3

Partial Evaluation

3.1 Introduction

Partial evaluation has been explored for a variety of languages ranging from functional to logic languages. In recent years, partial evaluators for real-sized languages such as Fortran [12] and C [6, 41] have been developed. Realistic applications of partial evaluators to various areas like systems software [116] and scientific computing [12] have clearly demonstrated that partial evaluation is an effective tool to allow programmers to write generic programs without loss of efficiency.

Intuitively, partial evaluation is aimed at instantiating a program with respect to some of its parameters. By restricting a generic program to a specific usage context, we hope to enable aggressive optimizations. Partial evaluation performs aggressive interprocedural constant propagation (of all data types). Based on the input values specified by the user and on constants explicit in the program, partial evaluation performs constant folding, as well as optimizations such as function unfolding, loop unrolling and some forms of strength reduction. Partial evaluation differs from ordinary optimization in that no resource limits are imposed on the computations that are performed during transformation. While this enables transformations that are out of the scope of an ordinary compiler, it does imply that the partial evaluation process must be guided by the user.

Partial evaluation is usually seen as a source-to-source transformation. Generating specialized programs in source form facilitates inspection to verify the quality of the specialized program. However, the set of values to specialize over may become gradually available during compilation and execution. Thus, a program may need to be specialized both at compile time and at run time. In fact, the partial evaluator Tempo offers both strategies in a uniform environment. Its run-time specialization capability has shown to produce efficient programs and to be amortized in a few runs of the specialized program [43]. Run-time specialization widens the opportunities to eliminate genericity in programs.

Although object-oriented languages encourage genericity, and thus offer opportunities for specialization, partial evaluation has so far been mostly unexplored for this language paradigm. Object-oriented languages are in some ways more complex than functional and imperative languages, and in this respect represent a natural progression for partial evaluation from functional and imperative languages. Interestingly, several forms of specialization have already been defined for object-oriented languages [33, 48, 155, 162]; we will present these concepts before we define partial evaluation for object-oriented languages.

The purpose of this chapter is to provide an overview of partial evaluation and a discussion of existing ideas about specialization for object-oriented languages. We cover those partial evaluation concepts that are essential for understanding the later presentation of partial evaluation for object-oriented languages. In addition, we evaluate existing ideas about specialization for object-oriented languages with regards to their relevance to partial evaluation.

Chapter overview: First, Section 3.2 introduces the basic principles of partial evaluation, and describes partial evaluation for functional and imperative languages. Then, Section 3.3 presents existing work and ideas about specialization of object-oriented programs, and last, Section 3.4 summarizes the contents of this chapter.

3.2 Principles of Partial Evaluation

Object-oriented languages are similar to functional and imperative languages in many aspects. Partial evaluation for functional and imperative languages has been investigated extensively in literature, which serves to provide a starting point for defining partial evaluation for object-oriented languages. Indeed, as we shall see in later chapters, partial evaluation for object-oriented languages borrows extensively both from partial evaluation for functional and imperative languages.

The logical language paradigm is in many ways close to the functional language paradigm. In fact, partial evaluation for logical languages is sufficiently close to partial evaluation for functional languages that it does not offer any additional insights into partial evaluation for object-oriented languages. At the same time, ideas in partial evaluation for functional languages are to a higher degree directly applicable to partial evaluation for object-oriented languages. For these reasons, we do not not treat partial evaluation for logical languages here; we refer the interested reader to the partial evaluation book [84] for more information about partial evaluation for logical languages.

In this section, we first describe partial evaluation extensionally, and then as a program transformation for functional and imperative languages.

3.2.1 Partial evaluation of a program

We can give an extensional characterization of the effect of partial evaluation on a program. Consider a program P that given two inputs s and d evaluates to a value v ($[\![\cdot]\!]$ indicates the evaluation function):

$$[\![P(s,d)]\!] = v$$

Kleene's S_n^m-theorem [92] ensures the existence of a program P_s specialized for the input s, such that:

$$[\![P_s(d)]\!] = [\![P(s,d)]\!] = v$$

The program P_s can be a trivial specialization of P, i.e., a program that simply calls P with s as a fixed argument. We are interested in the case where P_s is a simplified version of P, adapted to the fixed input s, that only makes computations and decisions depending on the remaining input d. A *partial evaluator* is a program *pe* that automatically computes P_s given P and s:

$$[\![pe(P,s)]\!] = P_s$$

The known information (s above) is referred to as *static*, whereas the unknown information is referred to as *dynamic*. Computations that can be reduced based on information available during specialization are said to have static *binding time* (or simply to be static), and all other computations have dynamic binding time (are dynamic). Partial evaluation eliminates static computations and reconstructs dynamic computations literally to form the specialized program.

Contemporary partial evaluators can be divided into two classes, on-line and off-line. An on-line partial evaluator is a non-standard interpreter that interprets a program by reducing static parts and residualizing dynamic parts; it decides on-the-fly whether to classify a program part as static or dynamic. An off-line partial evaluator is staged into two phases, the *binding-time analysis* and the specialization phase. The binding-time analysis calculates a *binding-time division*, which classifies each program part as being static or dynamic. A binding-time analysis that allows a single binding time per program part is said to be monovariant, whereas a binding-time analysis that allows multiple binding times per program part is said to be polyvariant. The binding-time division determines the actions of the specialization phase, which evaluates static program parts and residualizes dynamic program parts. Since the specialization actions are determined before specialization, a dedicated specializer can be generated given a specific binding-time division. In effect, the specialization actions and the static program parts can be compiled into a program generating residual programs, a so-called *generating extension*. This approach makes the specialization phase as efficient as standard compiled execution. Thus, off-line partial evaluation is more efficient than on-line partial evaluation. Conversely, on-line partial evaluation is more precise, because the specialization actions are determined on-the-fly, and thus can be based on the concrete values computed during specialization.

Throughout this document, we concentrate on off-line partial evaluation. We hypothesize that many of the specialization strategies and ideas that we develop for off-line partial evaluation of object-oriented programs apply to on-line partial evaluation. However, we consider on-line partial evaluation for object-oriented languages to be future work.

3.2.2 Partial evaluation for functional languages

A functional program can be viewed as a stateless mathematical function that describes a relation between input and output. Partial evaluation specializes a functional program to a static input by eliminating those parts of the relation that are fixed given the static input. This transformation simplifies the definition of how an input relates to an output, which reduces the amount of machine computation needed to map an input to an output.

In off-line partial evaluation, the binding-time analysis phase determines how the program should be transformed (specialized), and the specialization phase then performs the transformation. Thus, it is the binding-time analysis that should ensure that specialization does not "go wrong," i.e., by evaluating an expression that depends on unknown information. We first explain how a functional program can be specialized given a well-formed (i.e., safe) binding-time annotation, and then explain how such a binding-time annotation can be computed.

Specialization should reduce static program parts and residualize dynamic program parts. When an operation (e.g., addition) has been annotated static, it can be evaluated by the specializer. Evaluation is done by invoking the operation on the argument values, which yields a value available for further specialization. An operation annotated as dynamic is simply residualized along with its (specialized) arguments. A conditional expression that has a statically annotated test can be evaluated by the specializer. The evaluation of the test decides which branch to specialize. If the branches are annotated static, specialization of the conditional yields a value. Conversely, if the branches are dynamic, specialization of the conditional yields a residualized expression. A closure invocation that has been annotated static can be evaluated by unfolding its definition at the call site; this invocation consists of substituting any arguments throughout the body. (Unfolding is not essential: a residual call to a specialized version of the callee can also be generated, which enables sharing of specialized code between residualized call sites.)

The constraint that binding-time annotations should ensure safe specialization brings type inference to mind. Indeed, binding-time analysis for functional languages is easily understood in terms of a non-standard type inference system based on the classical algorithm $\mathcal{W}$ for polymorphic type inference [66, 112, 119]. Given type annotations on the parameters of the specialization entry point (the main program function to be specialized) with static and dynamic as base types, the entire program can be typed in terms of binding times. An operation that is passed arguments annotated as static is annotated static; it is annotated dynamic otherwise. A conditional is annotated with the binding times of its branches (which must have the same binding time). An invocation of a closure value with static binding time is given the binding time of its possible callees; each callee is analyzed with respect to the binding times of the arguments to the closure (a control-flow analysis [146] can determine the set of possible callees).

It is essential to the principle of partial evaluation that there is a mix of static and dynamic computations: specialization of a completely dynamic program is the identity transformation, and specialization of a completely static program is equivalent to standard evaluation. A dynamic computation can appear in a static context when its value is not needed by the static context (e.g., a static conditional with dynamic branches); the dynamic computation will be residualized in the context of the enclosing static computation. Con-

```
(define power (lambda (exp op neutral)
  (lambda (base)
    (letrec ((loop (lambda (e)
                     (if (zero? e 0)
                         neutral
                         (op base (loop (1- e)))))))
      (loop exp)))))
```

Figure 3.1: Generic power program in Scheme.

versely, a static computation that appears in a dynamic context must somehow be residualized into this dynamic context. To allow a static computation to appear in a dynamic context, a *lift* operator is used. In terms of binding times, a lift operator converts (lifts) a static binding time into a dynamic one. It is the binding-time analysis that inserts lift operators, while keeping as much as possible of the program as static. In terms of specialization, the lift operator residualizes the representation of a static value into the program text. In most partial evaluators, only base values can be residualized, since there is no residual representation for a complex computed value. The lift operator enables *speculative evaluation* of the branches of a dynamic conditional: even though only one branch will be evaluated in a standard evaluation of the conditional, partial evaluation can speculatively specialize both branches, which yields a more specialized residual program.

Example

As an example of partial evaluation for functional languages, we revisit the power example of the previous chapter (Figures 2.1 and 2.2) in a functional setting. Consider the Scheme function shown in Figure 3.1. The `power` function computes a number of applications of an arbitrary binary operator onto a base value; the neutral value defines the value of the base case. The expression `((power y * 1) x)` computes x^y. (Scheme offers built-in first-order functions such as binary operators.) Depending on what is known about its arguments, we can specialize `power` in a number of different ways.

Suppose that the exponent (the `exp` parameter) is static. It is the value of the `exp` parameter that controls the recursive calls in the local `loop` function and the conditional. As a result, specializing the `power` function for a specific value of `exp` unfolds all recursive calls to `loop` and reduces away all tests. Specialization for the exponent `3` is shown with the function `power_3` in Figure 3.2. We can also specialize the `power` function for the values of the operator and the neutral value (say, `+` and `0`, with the exponent still `3`). This way we obtain the specialized function `power_3_+_0`, which can be optimized into the function `power_3_+_0_opt` using arithmetic simplifications (both are shown in Figure 3.2).

```
; exponent known
(define power_3 (lambda (op neutral)
  (lambda (base)
    (op base (op base (op base neutral))))))

; exponent, operator and neutral known
(define power_3_+_0 (lambda (base)
  (+ base (+ base (+ base 0)))))

; ... and optimized
(define power_3_+_0_opt (lambda (base)
  (+ base base base)))
```

Figure 3.2: Specialized versions of power program of Figure 3.1.

3.2.3 Partial evaluation for imperative languages

An imperative program can be regarded as an ordered sequence of instructions that manipulate program state. Partial evaluation specializes an imperative program by eliminating the state manipulation that is fixed given the static input and simplifying the state manipulation that depends partly on static input. This transformation reduces the total amount of state manipulation performed during a run of the program.

The main complication when performing partial evaluation of an imperative program is side-effects. The order of computation must be respected during specialization. Interprocedural specialization should be depth-first to preserve the order of side-effects. When performing speculative specialization of a conditional, both branches must be evaluated in the same store. Thus, the specializer must save the program store before evaluating one branch, and then restore it for evaluating the other branch.

To correctly determine the flow of information through the program, the binding-time analysis must take side-effects into account when determining the binding time of each computation. An alias analysis [57, 164] can be used to annotate each statement of the program with precise information regarding what locations are read or written. The binding-time analysis is easily understood as an abstract interpretation of the program, with binding-time properties as values that can be stored in and read from the locations determined by the alias analysis. Side-effects made speculatively under dynamic control (i.e., inside a dynamic conditional) cause the side-effected locations to become dynamic.

Procedure calls are usually first order, hence the callee is known at the call site, and no control-flow analysis is needed. As for loops, a loop can be considered static when the termination condition is static. It can be specialized by unrolling the body according to the termination condition, with each version of the body specialized to the current program store.

```
int add( int i1, int i2 ) { return i1+i2; }
int mul( int i1, int i2 ) { return i1*i2; }
int power( int exp, int (*op)(int,int), int neutral, int base ) {
  int res = neutral;
  while( exp-- ) res = (*op)( res, base );
  return res;
}
```

Figure 3.3: Generic power program in C.

Example

As an example of partial evaluation for imperative languages, consider the C functions shown in Figure 3.3. The C `power` function corresponds to the Scheme `power` function used to illustrate partial evaluation for functional languages. We have defined auxiliary functions encapsulating binary operators, so that we can pass them as arguments and apply them using a C function pointer. Again, depending on what is known about the arguments, we can specialize `power` in a number of different ways.

Suppose that the exponent (the `exp` parameter) is static. It is the value of the `exp` parameter that controls the iteration in the `while`-loop. Specialization for a specific value of `exp` unrolls the loop, as shown for the exponent 3 with the specialized function `power_3` in Figure 3.4.[1] In addition to the exponent, we can also specialize the `power` function for the values of the operator and the neutral value (say, a pointer to the function `add` and `0`, with the exponent still `3`). This way we obtain the specialized function `power_3_add_0`; optimizing `power_3_add_0` yields the specialized function `power_3_add_0_opt` (both are shown in Figure 3.4).

3.3 Specialization for Object-Oriented Languages

The notion of specialization is very general; partial evaluation is one form of program specialization, as is deforestation [163] for example. Before we define partial evaluation for object-oriented languages, we compare partial evaluation to existing forms of specialization for object-oriented languages. We investigate the relevance of inheritance to partial evaluation, and likewise the relevance of class hierarchy specialization to partial evaluation. Then, we compare customization and selective argument specialization with partial evaluation, and discuss an existing proposal for declaring specialization of object-oriented programs.

[1]A partial evaluator like Tempo [41] handles function pointers, but not quite as intelligently as in this example, since the function pointer calls would have been residualized as conditionals.

```
/* exponent known */
int power_3( int base, int (*op)(int,int), int neutral ) {
  int res = neutral;
  res = (*op)( res, base );
  res = (*op)( res, base );
  res = (*op)( res, base );
  return res;
}

/* exponent, operator and neutral known */
int power_3_add_0( int base ) {
  int res;
  res = add( 0, base );
  res = add( res, base );
  res = add( res, base );
  return res;
}

/* ... and optimized */
int power_3_add_0_opt( int base ) {
  return base+base+base;
}
```

Figure 3.4: Specialized versions of power program of Figure 3.3.

3.3.1 Inheritance

Many object-oriented languages have an intrinsic notion of specialization, in the form of inheritance between classes. To the extent that we can compare a programming construct and an optimization technique, it seems that partial evaluation is the inverse of inheritance. Inheritance adds new state, whereas partial evaluation fixes static state in the program. Inheritance makes it possible to add new behavior, whereas partial evaluation only can simplify behavior.

A subclass is said to specialize its superclass; it can refine the behavior of methods in the superclass and add new behavior and state to behave in a more specific way. On the contrary, partial evaluation simplifies behavior by removing computations that depend on a fixed state. Intuitively, it then seems possible that partial evaluation could be a means of deriving superclasses of a given class. However, this idea does not integrate well with inheritance: the derived superclasses would each duplicate the generic parts of the original class, and each duplicated part would be inherited separately back into the original class (a classical multiple inheritance problem). Besides, partial evaluation is usually viewed as specializing an entire program into a new program, whereas inheritance only specializes a single class into a new class (or a collection of classes when using multiple inheritance). We thus choose to dissociate the notions of partial evaluation and object-oriented inheritance; although partial evaluation and inheritance may interact, they fulfill different purposes.

3.3.2 Class hierarchy specialization

When class definitions are reused across different programs, it may often be the case that the class definitions are more general than is required for a specific program. Tip and Sweeney specialize a class hierarchy to a specific program by creating a new distinct set of classes where members not needed for specific objects in a given program are removed [155]; their technique is referred to as *class hierarchy specialization*.

Class hierarchy specialization is a form of program specialization for object-oriented languages, targeted towards specializing the data representation of a program. The program may be transformed, but only where it serves to optimize the data representation. Conversely, partial evaluation for functional and imperative languages is targeted towards reducing the amount of computation in the program. The program data representation may be changed, but only where it serves to optimize execution speed. Consider the following two examples. A partial evaluator for a functional language may move dynamic values from a static closure to become formal parameters using *arity raising* [139]. The C-Mix partial evaluator for the imperative language C replaces dynamic memory allocation with static memory allocation when possible [6]. Although useful, none of these transformations are essential to the nature of partial evaluation for functional and imperative languages.

We conclude that a partial evaluator for an object-oriented language could perform different forms of class hierarchy specialization depending on the static data, and that this would probably optimize the overall performance of the program. However, we do not consider the idea of class hierarchy specialization to be essential to the nature of partial evaluation for object-oriented languages. We will return to class hierarchy specialization later as a direction for future work (in Chapter 13).

3.3.3 Customization and selective argument specialization

Object-oriented mechanisms such as inheritance and virtual dispatching offer a well-structured approach to building generic programs from reusable program parts. Chambers and Ungar specialize object-oriented programs using *customization*, which generates multiple variants of commonly used methods, each one specialized for a specific potential type of the receiver object [33]. The selection of what specialized method to use is done implicitly using the virtual call mechanism; the type of the receiver object decides what method to use. A method is specialized by using the known type information to eliminate virtual dispatches and generate specialized variants of callee methods. Customization will normally specialize all methods in a class, which can lead to overspecialization. At the same time, customization only specializes for the type of the self object, which can lead to underspecialization when the type of some other argument is important for optimizing the program. As a solution, Dean, Chambers and Grove present a generalization of customization that specializes methods for any of their arguments, but concentrates on specializing methods for those method arguments that allow significant optimizations to be performed [48]; this specialization technique is called *selective argument specialization*. Selective argument specialization relies on multi-dispatching [19] in the execution model to select between method variants specialized for arguments other than the self object. To avoid code explosion, selective argument specialization normally relies on profile information to detect critical program parts where specialization is applied (see Chapter 12 for more details).

Selective argument specialization provides a useful indication of how partial evaluation could transform an object-oriented program, namely by specializing its methods. However, selective argument specialization does not specialize methods for base-type values, whereas partial evaluation should specialize for all kinds of values. In addition, multi-methods are needed in the target language to express dispatching to methods specialized for arguments other than the self object. Partial evaluation is a source-to-source transformation, and can therefore not rely on the presence of multi-methods in the target language (not all object-oriented languages have multi-methods). At a more fundamental level, selective argument specialization relies on an analysis to determine a set of types for which to specialize methods, whereas partial evaluation relies on program execution to obtain the concrete value for which a method is specialized. Also, partial evaluation and selective argument specialization differ in their degree of automation. Partial evaluation has the benefit of being controlled by the user to target a specific specialization opportunity, and performs highly aggressive optimizations without resource bounds. Selective argument specialization is a general-purpose optimization technique that operates without the need for user control, which inevitably imposes bounds on the amount of computation that can be used to specialize a program. Thus, we consider selective argument specialization a source of inspiration, and will return to it later as a compiler optimization when considering performance issues (Chapter 11) and in terms of its relation to partial evaluation for object-oriented languages (Chapter 12).

3.3.4 Specialization classes

Partial evaluation relies on user guidance to describe what program part to specialize for which usage context. Nevertheless, partial evaluators usually offer an intricate interface to the user, which complicates the task of specializing a program (the Tempo specializer for C is a good example [41, 133]). As a means of circumventing this problem, Volanschi et al. present a declarative framework that offers a unified means of indicating what program part to specialize for which usage context, and enables automatic reintegration of specialized code into the program [162]. This framework, named *specialization classes*, is targeted towards certain opportunities for specialization found in object-oriented programs, and hence offers a perspective on the nature of partial evaluation for object-oriented languages.

Specialization classes support the specialization strategy of customization and selective argument specialization: the methods of the program are specialized to known values. However, specialization classes were developed before partial evaluation for object-oriented languages was developed. Specialization classes are mostly concerned with objects as a means of encapsulating imperative methods and the data that they operate on. They offer only limited support for expressing information about the interaction that takes place between objects, which we consider essential information when specializing an object-oriented program. For this reason, we will define partial evaluation for object-oriented languages with specialization classes in mind, but independently of the current incarnation of specialization classes. In Chapter 6 we will return to specialization classes, and extend them with the capabilities that we require.

3.4 Summary

Partial evaluation is an automatic technique for specializing a program to a given usage context. Specialization reduces away all computations that only depend on static (known) information, and leaves behind those computations that depend on dynamic (unknown) information. We concentrate on off-line partial evaluation, which is divided into two phases. The binding-time analysis annotates each computation of the program as being either static or dynamic. The specialization phase evaluates the static computations and residualizes the dynamic computations. Partial evaluation for functional languages incorporates most of the basic concepts of partial evaluation; the complications in dealing with imperative languages stems from the presence of side-effects.

In terms of existing approaches to specialization for object-oriented languages, selective argument specialization and specialization classes offer a simple perspective on specialization of object-oriented programs: methods are specialized to known data. We keep these ideas in mind for the next part of the document, where we investigate how partial evaluation for object-oriented languages can be made a useful software engineering technique.

Part II

Software Engineering and Partial Evaluation

Chapter 4

Software Engineering

4.1 Introduction

Ideally, partial evaluation can specialize a generic program when given a static input that fixes part of the program's behavior. In practice, partial evaluators are limited by the precision of their static analyses. The more features are added to a static analysis to enhance its precision, the more complex and thereby unpredictable the analysis. In defining partial evaluation for object-oriented languages, we will aim for a more modest goal than specialization of arbitrary generic programs: our partial evaluator should be capable of specializing programs that have been written with specialization in mind.

For a partial evaluator to be a useful object-oriented programming tool, it must work with and complement object-oriented development methodology. Object-oriented software development is a well-researched domain, that covers most aspects of solving a given information processing problem. However, the primary effort is towards researching ways of developing general solutions to problems, as opposed to developing efficient solutions. To reconcile object-oriented software development and efficiency, we propose to integrate partial evaluation into the object-oriented software development process, and to use it to automatically map general object-oriented solutions into efficient ones.

This chapter presents software engineering issues related to partial evaluation and object-oriented languages. We outline how partial evaluation has been used as a software engineering tool to improve program execution efficiency, and evaluate to what degree partial evaluation has proven to be a useful tool. We conclude that partial evaluation must take the software engineering process into account, and investigate how partial evaluation can complement object-oriented software engineering principles.

Chapter overview: First, Section 4.2 presents partial evaluation as a software engineering tool, and Section 4.3 assesses its overall success as a programming tool. Then, Sections 4.4 and 4.5 discuss the object-oriented approach to system design and the concept of design patterns, both in light of efficiency issues and partial evaluation. Last, Section 4.6 presents a summary of this chapter.

4.2 Software Engineering using Partial Evaluation

Partial evaluation has been demonstrated to be a powerful tool for eliminating overheads in generic programs: it automatically derives an efficient implementation for a given usage context. This section provides an overview of some realistic examples of applying partial evaluation to obtain efficient implementations; we investigate three different domains, namely scientific computations and computer graphics, operating systems, and software architectures.

4.2.1 Scientific computations and computer graphics

Scientific computations and computer-graphics are often computationally intensive, are naturally parameterized, and usually require efficient solutions due to time constraints. These factors make them interesting targets for partial evaluation. The data structures of a scientific computation program are often shaped according to a conceptual understanding of the problem being solved. Berlin observes that the configuration of a data structure often remains invariant between computations. He demonstrates how partial evaluation can be applied to Lisp programs to transform computations over data structure elements into operations applied directly to the variable program inputs [14]. In general, scientific computations are often parameterized by multiple values, some of which remain invariant when solving the same problem repeatedly for different data sets. Baier, Glück and Zöchling show how their partial evaluator for Fortran can specialize realistic library routines for invariant parameters to obtain more efficient routines [12]. Günter, Knoblock and Ruf attack the specific problem of shaders for three-dimensional image rendering, embodied by an application that allows the user to interactively adjust various shading parameters for a fixed scene. Here, it is often the case that some scene parameters (surface properties and lightsource position, for example) remain invariant; precomputing intermediate results that depend only on invariant parameters can greatly speed up the rendering process [70].[1]

4.2.2 Operating systems

Operating systems are not as computationally intensive as scientific computations, but nonetheless still contain many opportunities for specialization. Indeed, many subsystems are designed in a generic and parameterized way. Furthermore, interpretation and traversal of data structures can also be an overhead; Consel et al. present a partial evaluator for C designed towards specializing such generic operating system components [41]. This partial evaluator, named Tempo, incorporates a range of state-of-the-art static analyses and aggressive transformations developed to handle the specialization opportunities found in operating system components. Engler and Kaashoek observe that programmable packet filters interpret a complex set of rules every time they

[1]Due in part to the massive size of the data sets for which specialization is done, and in part to the interactive nature of the system being specialized, computed static values are advantageously saved in a cache rather than residualized into the program, using a partial evaluation technique called *data specialization* [94].

receive a packet. They exploit dynamic code generation to generate dedicated packet filters for a set of rules [59]. Muller et al. apply partial evaluation to optimize the Sun RPC implementation; they simplify computations over a partially static data structure containing control options, and specialize argument and safety checking routines [116]. Operating system components usually follow a standard pattern of design where error-checking is done based on the exit status of subroutines. Muller et al. exploit this recurring specialization opportunity by propagating the subroutine exit status; specialization eliminates the error-checking code when it is not needed.

4.2.3 Software architectures

Software architectures offer a global approach to organizing programs, by working within a framework [145]. Frameworks help structure large programs built from smaller components, which improves genericity and adaptability in the design of a system. However, genericity and adaptability are present not only when structuring a software system, but also in its implementation: data exchanged between components may need to be converted to an appropriate format, and the control flow of individual independent components must be integrated. Marlet et al. demonstrate how partial evaluation can be systematically applied to eliminate the architectural overheads of software architectures, and used to automatically generate an efficient implementation [106].

4.3 Assessment of Partial Evaluation

Clearly, partial evaluation has been applied with success to optimize realistic programs. Partial evaluation can instantiate a parameterized program, tailoring it to implement only a specific functionality. The interpretive overhead due to testing of and computing over static parameters is removed, reconciling the otherwise opposing needs of genericity and efficiency. Nonetheless, partial evaluation has never gained wide acceptance as a programming tool. Indeed, it is difficult to use partial evaluation to specialize a program that is not appropriately designed; for partial evaluation to work, any opportunities for specialization intrinsic to a program should be explicit in its structure. For this reason, we propose that partial evaluation must be taken into account in the design of the program.

4.3.1 Partial evaluation pitfalls

The practical experience of the Compose group in specializing C and Java programs has demonstrated that specializing an existing program using state-of-the-art partial evaluation technology often requires a significant effort in terms of adapting the program to specialize well. The difficulty lies in assuring that the automatically derived binding times match the desired degree of specialization; too little specialization makes partial evaluation uninteresting, and too much specialization can cause code explosion. Non-termination of the specialization process can also be caused by improper binding times. However, non-termination is less of a problem in practical use, and is well-investigated in literature (see for example [84, Chapter 14]).

To ensure appropriate binding times, the partial evaluation user must understand the structure of the program, and have an idea as to what binding times are needed at critical program points. When a binding time does not match the expectations, a manual code transformation is normally performed, known as a *binding-time improvement*. Program-independent binding-time improvements such as *bounded static variation* a.k.a. "the trick" [84] are well known in the partial evaluation community. Program-specific binding time improving transformations are also described in literature, for example for specializing interpreters [83, 154]. Nonetheless, the effort needed to adapt an existing program to specialization may at times surpass the effort needed to manually specialize the program or to come up with an alternative and more efficient program.

4.3.2 Partial evaluation and software development

To circumvent the difficulties in applying partial evaluation, we propose to limit its scope of applicability. Rather than applying partial evaluation to optimize an existing program, the program should be developed with partial evaluation in mind; this approach ensures that the program will specialize well — given that it is well written. As a program part is being developed, it should be verified by the programmer that it can specialize well. To this end, partial evaluation should be part of the software development process, and integrated into the software engineering methodology. In the particular case of partial evaluation for object-oriented languages, partial evaluation should work with and complement existing techniques in object-oriented software engineering. The decomposition of a problem into objects should be reflect how partial evaluation is applied to the program, and standard object-oriented designs should be analyzed for how they can be optimized using partial evaluation.

4.4 Object-Oriented Software Development

Object-oriented software engineering is a rich and well-researched domain, that includes all aspects ranging from the initial analysis of a system to creating a concrete implementation. There are numerous aspects of object-oriented software engineering which are interesting in conjunction with partial evaluation, many more than can be covered in this document (see the discussion of future work in Chapter 13 for more details). We here concentrate on programming language aspects of an object-oriented implementation constructed straightforwardly based on standard object-oriented principles.

In this section, we first present an overview of object-oriented program development, then investigate the overheads that are intrinsic to programs written according to standard object-oriented development principles, and finally discuss how partial evaluation can be applied to eliminate these overheads.

4.4.1 Developing object-oriented programs

Object-oriented analysis and design is concerned with mapping a part of reality (the referent system) into a computerized physical model [102]. An analysis identifies the relevant elements of the referent system, and an abstraction process yields an initial object-oriented design of the computerized system (typically in the form of object diagrams). The design is then translated into the abstractions offered by an object-oriented programming language, and the implementation of each model part yields a complete program.

The direct mapping of the referent system through an abstract design into an implementation ensures that any intrinsic potential for adaptation in the referent system is preserved in the implementation. The decomposition of the implementation into autonomous objects that mirror similar entities in the referent system ensures easy extension of the model to match additional capabilities in the referent system determined after the initial design phase.

4.4.2 Common overheads in object-oriented programs

For the advantages ordinarily associated with object-oriented programming to apply, a solution to a problem must be decomposed into objects. The more fine-grained and detailed the modeling of a system, the more fine-grained this decomposition of the problem into objects. However, decomposition into numerous objects carries a dual performance disadvantage. First, access to values stored in object fields becomes more expensive due to the amount of data structure traversal needed to obtain actual values (one dereferencing operation per object nesting). Second, fine-grained objects imply many smaller methods, that interact often.

Objects interact using method invocations, which can be implemented either as direct calls or virtual calls. Virtual calls defeat branch prediction and thereby instruction pipelining. They also inhibit inlining, which blocks subsequent traditional intraprocedural compiler optimizations [28, 52]. For these reasons, many compilers go to great lengths to replace virtual calls by direct calls [3, 50, 51, 130]. Some compilers even make use of constrained specialization techniques, such as customization and selective argument specialization [33, 48]. Even so, virtual calls can only be completely eliminated when a static analysis can safely determine that the class of the receiver object never changes. Nonetheless, a virtual call can be replaced with an explicit selection of which callee to invoke with a direct call [69, 76]. This highly aggressive optimization must be guided with profile information to avoid code explosion, and retains the cost of a runtime decision.

4.4.3 Using partial evaluation

The decomposition of a system into many small objects provides numerous advantages in terms of software development, but may be more fine-grained than is needed for a specific application. Given information about what parts of the decomposition is fixed, partial evaluation can fix the relations between individual objects. Only the needed decomposition points remain as sources of inefficiency.

As part of the analysis, design and implementation of an object-oriented system, potential fixed parts of the object decomposition can be identified. This information can then be used later to specialize the system. Chapter 6 elaborates further on how the application of partial evaluation to a program can evolve with program development.

4.5 Design Patterns

Design patterns, as presented by Gamma et al. [64], describe well-tested program structures that enhance modularity and code reuse. A program written using design patterns is structured into independent units that interact through generic interfaces, and that can evolve over time. Because design patterns are well-documented, their use simplifies the understanding of programs constructed from many independent units. The flexibility inherent in this use of generic interfaces, however, intrinsically blocks optimization across objects, and thus can carry a significant performance penalty.

In this section, we first present an overview of design patterns, then investigate the overheads that are intrinsic to programs written using design patterns, and finally discuss how partial evaluation can be applied to eliminate these overheads.

4.5.1 Capturing object-oriented designs

A design pattern systematically names, explains, and evaluates an important and recurring design in object-oriented systems; it encapsulates a useful design idea, and allows it to be communicated in the object-oriented community. The goal of a design pattern is to capture design experience in a form that people can use effectively. A design pattern is composed of a name, the problem that it addresses, the solution that it proposes, and the consequences of applying the pattern [64]. Associating a name with the pattern allows problems and solutions to be described in terms of the pattern, which facilitates communication of design ideas. Identification of when the pattern is useful is eased by a precise description of the target problem. Use of the solution proposed by the pattern results in a program structured according to well-tested design principles. Awareness of the consequences of choosing a specific solution helps to minimize subsequent problems in the design.

The collection of design patterns presented by Gamma et al. [64] focus on writing adaptable object-oriented programs; an adaptable program component can be used for multiple purposes both during program development and during the execution of a program. Inheritance and delegation are fundamental to writing adaptable object-oriented programs, but the use of inheritance and delegation obscures the relationship between program components. Design patterns clarify program structure by documenting such component relationships. Each design pattern focuses on a single problem often encountered in developing an adaptable program and proposes a widely applicable solution. By using a design pattern the programmer takes advantage of a well-tested program structure that is open for future extensions. Adaptable programs can be described

```
class Set {
  MinimalCollection coll;            // underlying collection
  boolean member( Object o ) {
    Iterator e = coll.iterator();    // obtain iterator
    while( e.hasNext() ) {           // while iterator has elements
      Object x = e.next();           // obtain next iterator element
      if( x.equals( o ) ) return true;
    }
    return false;
  }
  ...
}
```

Figure 4.1: A standard use of the iterator design pattern.

in terms of the design patterns they implement. The design patterns provide a guide as to how the functionality of the program is likely to be distributed among the class definitions.

4.5.2 Overheads from using design patterns

Most design patterns distribute functionality among objects that cooperate through abstract interfaces, which reduces software adaptation to a reorganization of the objects that constitute the program. The potential to reorganize the program objects at any moment implies that most interaction between objects must take place using virtual calls. However, design patterns may often provide more adaptability than is needed within a specific phase or run of a program.

As a simple example, consider how the traversal of a data structure can be separated from its representation using the *iterator* design pattern [64]. With the iterator pattern, an implementation of a set data structure (named `Set`) can define the member operation (the `member` method) as shown in Figure 4.1. This definition of the `member` method can be used with any underlying implementation of the `MinimalCollection` interface, letting the programmer freely choose the most appropriate concrete representation for the task at hand. Nevertheless, the experiments reported later in this document show that the use of the iterator pattern blocks compiler optimizations of the element retrieval operations. When the `member` method is used repeatedly to search `MinimalCollection` objects that have the same representation, the flexibility provided by accessing the data through an abstract interface is not needed. Replacing the generic uses of the iterator pattern (underlined in the method definition) by direct accesses to the underlying data structure gives a speedup ranging from 20% to 80%.[2]

[2]More detailed experiments are documented in Chapter 11. They are done using state-of-the-art Java compiler technology on a program with array and linked list representations of the underlying `MinimalCollection` data structure; the array data structure is shown in Figure A.1 in of the appendix.

These measurements suggest that the optimizations performed by state-of-the-art compilers do not completely compensate for the genericity introduced by design patterns.

When a design pattern is used in a performance-critical part of the program, it is essential for the performance of the program that any virtual dispatch induced by the design pattern use be optimized. Standard optimization techniques might often fail, since the virtual dispatch forms a point of adaptation that potentially might reach numerous callees. Specialization techniques come to mind as a solution, but for such techniques to be effective, they must only be applied to the performance-critical parts of a program. In the case of automatic compilation systems, profiling is used to determine the performance-critical program parts. However, overheads that are distributed evenly throughout a program (and thus not restricted to any local region) are more difficult to detect. The structural overheads that are introduced by uses of design patterns fit into this category, which makes them difficult to optimize.

4.5.3 Using partial evaluation

To simplify program development and facilitate communication between program developers, programs may be explicitly organized according to well-known design patterns, even when the full flexibility provided by the chosen design patterns is not needed. Partial evaluation can be applied to remove program flexibility not needed in a specific usage context, which reduces the number of computationally expensive adaptation points in the program. The fact that design patterns are well-documented and can be recognized within the code makes them easier to handle; the design patterns are themselves the key to accurately targeting the specialization process to these structural overheads. We return to this point in Chapter 7, after having investigated partial evaluation for object-oriented languages in more detail.

4.6 Summary

Partial evaluation is a powerful software engineering tool that can automatically map a generic, parameterized program into a specific, concrete implementation. Partial evaluation has been demonstrated to be applicable within the fields of scientific computations, computer graphics, operating systems, and software architectures. Nonetheless, practical experience shows that applying partial evaluation to a finished program is a complicated task. To overcome this limitation, we propose to consider partial evaluation as part of the software engineering process: a program that is to be specialized using partial evaluation should, from the initial conception to the final testing, be developed with partial evaluation in mind.

When following the standard object-oriented software development approach, a solution is implemented in terms of objects that reflect the problem domain. This decomposition into objects lies at the base of the software engineering advantages commonly associated with programming in object-oriented languages, but introduces overheads into the final program. Partial evaluation can be applied to eliminate these overheads, by supplying information to the partial evaluator about fixed parts of the object decomposition. Design patterns describe well-tested program designs that can be used in program design to easily attain the benefits associated with well-tested program structuring mechanisms. Many design patterns enhance the adaptability of a program, which directly improves code reuse, but also carries a performance overhead. The partial evaluator can be fed information about what adaptation is needed in the program and then used to eliminate such overhead.

In effect, we propose that programs should be written according to standard object-oriented development methodology *but* with partial evaluation in mind, and then mapped into efficient implementations using partial evaluation. In doing so, object-oriented principles can be used even when unacceptable performance overheads would normally be introduced in the final program. The opportunities for specialization made explicit in the program are exploited using partial evaluation as the final software development phase; this last step ensures an efficient implementation obtained through automatic specialization.

Chapter 5

Object-Oriented Partial Evaluation

5.1 Introduction

The object-oriented style of programming facilitates program adaptation. Encapsulation enhances code resilience to program modifications and improves the possibilities for direct code reuse. Message passing lets program components communicate without relying on a specific implementation; this decoupling enables dynamic modification of the structure of the program to react to changing conditions. However, in those cases where a more static model would have been sufficient, this potential for adaptability is achieved at the expense of efficiency. We are interested in partial evaluation as a means of reconciling adaptability and efficiency. Partial evaluation is known to transform generic, adaptable programs into specific, efficient ones. In the case of object-oriented languages, partial evaluation should facilitate the development of object-oriented software and eliminate inefficiencies common to object-oriented programs.

We intend for partial evaluation for object-oriented languages, referred to as *object-oriented partial evaluation*, to be used as a software engineering tool, that can be applied by the programmer during software development to perform automatic specialization. Object-oriented programming methodology provides a systematic mapping of real-world problems into adaptive programs. Partial evaluation should complement this mapping by specializing such an adaptive program into an efficient implementation tailored to the specific problem at hand. To this end, object-oriented partial evaluation should specialize genericity expressed in terms of object-oriented abstractions, as well as generic imperative or functional program parts. The notion of encapsulation of data and actions into an object must be respected in order to preserve the object-oriented structure of the program.

The purpose of this chapter is to provide a conceptual account of how partial evaluation specializes a program. We define the global effect of partial evaluation as a simplification of the interaction that takes place between the objects constituting a program. This effect is achieved by specializing object-oriented mechanisms in conjunction with other program parts. Given an execution con-

```
class Binary extends Object {
  Binary() { super(); }
  int eval( int x, int y ) {
    return this.eval( x, y );
  }
}
class Add extends Binary {
  Add() { super(); }
  int eval( int x, int y ) {
    return x+y;
  }
}
class Mult extends Binary {
  Mult() { super(); }
  int eval( int x, int y ) {
    return x*y;
  }
}
```

```
class Power extends Object {
  int exp; Binary op; int neutral;
  Power( int exp, Binary op, int neutral ) {
    super();
    this.exp = exp;
    this.op = op;
    this.neutral = neutral;
  }
  int raise( int base ) {
    return loop( base, exp );
  }
  int loop( int base, int e ) {
    return e==0
     ? this.neutral
     : this.op.eval( base,
                     this.loop( base, e-1 ) );
  }
}
```

Figure 5.1: The power example (again).

text, the result of specializing an object-oriented program is a specific program aspect adapted to this execution context.

Chapter overview: Section 5.2 defines how partial evaluation specializes an object-oriented program, and Section 5.3 describes how to express the residual program using aspect-oriented programming. Then, Section 5.4 presents examples illustrating our approach, and Section 5.5 provides a summary.

5.2 Object-Oriented Partial Evaluation

Object-oriented partial evaluation is at its basic level more complicated than partial evaluation for functional or imperative languages, since there are complex abstractions to take into account in object-oriented languages. To explain partial evaluation for object-oriented languages, we first describe how partial evaluation transforms a program from a global point of view, and then from a local point of view. We then discuss speculative specialization for object-oriented partial evaluation, and afterward address how to express the result of specializing an object-oriented program. Last, we present a small example illustrating partial evaluation for object-oriented languages.

5.2.1 Specializing an object-oriented program

From a global point of view, the execution of an object-oriented program can be seen as a sequence of interactions between the objects that constitute the program. Fixing particular program input parameters can fix certain parts of this interaction. The purpose of partial evaluation should be to simplify the object interaction as much as possible, by evaluating the static (known) interactions, leaving behind only the dynamic (unknown) interactions.

As an example, let us revisit the power example with arbitrary binary operator and base case value, shown in Figure 5.1. This example was initially shown

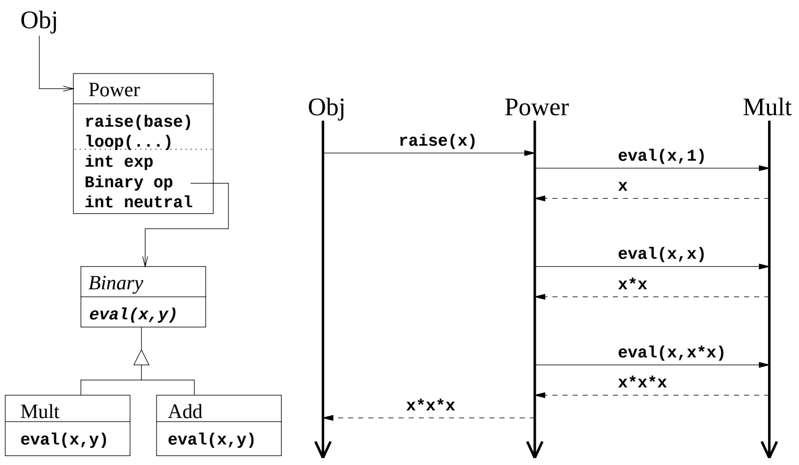


Figure 5.2: Generic interaction with `Power` object.

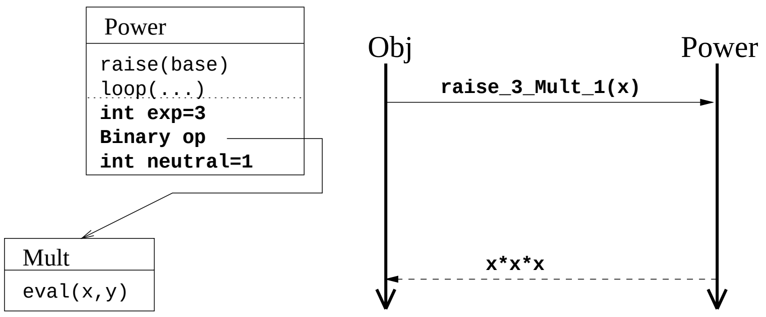


Figure 5.3: Specialization invariant and resulting interaction.

in Java (EFJ) in Chapter 2, and then revisited in Chapter 3 with equivalent Scheme and C versions, which were specialized. Figure 5.2 shows the class diagram of the power example (left) and the interaction diagram for computing x^3 (right). The interaction diagram shows how some other object `Obj` interacts with `Power` and indirectly with `Mult`: sending the message `raise(x)` to `Power` initiates a sequence of `eval` message sends from `Power` to `Mult`, which yields the result `x` $*$ `x` $*$ `x`.

We can specialize this program with respect to the information that the exponent value is `3`, the binary operator has type `Mult`, and the neutral value is `1`. This information is illustrated in the object diagram of Figure 5.3 (left). The interactions between `Power` and its binary operator are statically controlled by this information, whereas the computed result is dynamic since the value being raised is dynamic. To exploit this information while respecting encapsulation, the `Power` object can be enriched with new specialized behavior. To this end, the `Power` object is extended to respond to a new kind of specialized message; objects that use `Power` under the conditions for which it has been specialized send this new message. The code that implements the specialized message is integrated within the object, and is thus evaluated in the same environment as the original message. Figure 5.3 (right) shows the interaction that results from specialization to the static information. The method `raise_3_Mult_1` has been added to the `Power` object; this method directly returns the result `x * x * x` without further interaction with other objects.

We can view the method `raise` of the `Power` object as being parameterized not only by its formal parameter (bound to `x` in the example) but also by its self object. We have in fact specialized this method to the static parts of its parameters: the self object contains the exponent value, the neutral value, and the reference to the `Binary` operator object. The parameters define the complete context for an invocation of the method `raise` with a given `Power` object; by specializing the `raise` method to its parameters and subsequently adding it to the `Power` object, we adapt the `Power` object to its context. We can thus specialize the object interaction in a program by partial evaluation of the individual methods; each method is specialized with respect to the static part of its arguments.

This proposal fits well with the considerations about specialization for object-oriented languages presented Chapter 3: the program is specialized by specializing individual methods for information about their arguments. Indeed, this simple scheme is the basic approach that we propose for object-oriented partial evaluation.

5.2.2 Specializing object-oriented mechanisms

Central to all object-oriented languages are the notions of encapsulation and message passing. Related data and methods that operate upon this data are encapsulated into objects. Objects can communicate using virtual dispatching to send messages to receiver only known by its interface. To specialize object-oriented programs, we must take these mechanisms into account.

Optimizing encapsulation: Data that are encapsulated inside an object can control computations elsewhere in the program. When the data are considered static by the specializer, they can be propagated to wherever they are used, and computations that depend on the data can be reduced. Not only are computations that depend on the data reduced away, but the computation that was needed to access the data is also removed (an indirect memory reference).

Optimizing virtual dispatching: The callee selection that implicitly takes place in a virtual dispatch can be seen as a decision over the type of the self object. If the self argument is statically known, the decision of which method to invoke can be made during specialization. The callee method can be specialized based on information about its calling context, and can either be added to the receiver object, or be unfolded into the caller. Eliminating the virtual dispatch eliminates an indirect jump, which in turn simplifies the control flow of the program, and improves compiler optimizations and any pipelining performed by the processor. Not all object-oriented languages allow direct method calls to be expressed; we will return to this issue in more detail in Chapter 9.

Other optimizations: Apart from reducing field lookup and virtual method invocation, specialization of a method can also perform transformations commonly specific to partial evaluation for functional and imperative languages. Transformations such as constant propagation (of all data types), constant folding, conditional reduction, and loop reduction are all useful in partial evaluation for an object-oriented language. Indeed, an object-oriented language such as Java [67] also consists of functional and imperative constructs (conditionals, loops); it is thus essential that partial evaluation handles these constructs as well. In the Java specialization experiments reported in Chapter 11, most specialization scenarios require treating a mix of object-oriented and imperative constructs.

5.2.3 Speculative specialization

A virtual dispatch is similar to a conditional: partial evaluation reduces a virtual dispatch with a static self object, just like it reduces a conditional with a static test. A conditional with a dynamic test can be specialized speculatively; each branch is evaluated under the assumption that it was chosen. Similarly, if the set of possible callees is known to the partial evaluator, then each possible callee can be specialized speculatively. Here, the self object is dynamic while specializing each callee, but specialization can be done for any other arguments. The result is a residual virtual dispatch with the set of specialized methods as possible callees.

Speculative evaluation is only safe when the partial evaluator can determine that each potential callee method can be safely specialized (evaluated) for any static arguments. In a statically typed language a safe set of potential callees can trivially be determined using type inference (the typing rules ensure safe evaluation of each potential callee), but there is no obvious way to do the same thing in a dynamically typed language. Without speculative specialization, specialization stops where a dynamic decision is encountered; this approach can be a major drawback when specializing a large program. Thus, we consider that speculative evaluation of virtual dispatches is as important as speculative evaluation of conditionals, and that it is essential for partial evaluation of statically typed object-oriented languages. We consider the development of techniques for safe speculative evaluation of virtual dispatches in dynamically typed languages to be future work.

5.2.4 Appearance of the specialized program

Partial evaluation is usually applied as a source-to-source transformation, with run-time specialization as a notable exception. Producing the specialized program in source form enables one to verify that the specialized program matches one's expectations. Partial evaluation of an object-oriented program produces a set of specialized methods, each of which must be present in a given object

```
// exponent known,
// base, operator, neutral unknown
int Power.raise_3( int base ) {
  return this.op.eval(
          base,
          this.op.eval(
           base,
           this.op.eval(
            base,
            this.neutral ) ) );
}
// exponent, operator and neutral known
// base unknown
int Power.raise_3_Mult_1( int base ) {
  return base * ( base * ( base * 1 ) );
}
// ... and optimized
int Power.raise_3_Mult_1_opt( int base ) {
  return base*base*base;
}
```

(a) Specialization of `Power`

```
// exponent and base known
// operator and neutral unknown
int Power.raise_3_2() {
  return this.op.eval_2(
          this.op.eval_2(
           this.op.eval_2(
            this.neutral ) ) );
int Binary.eval_2( int y ) {
  return this.eval_2( y );
}
int Add.eval_2( int y ) {
  return 2+y;
}
int Mult.eval_2( int y ) {
  return 2*y;
}
```

(b) Speculative specialization

Figure 5.4: Specialization of the power example.

at a specific program point. In a class-based language, a specialized method must be present in the class of the object instance that received the method invocation. In an object-based language, a specialized method must be present in the specific object instance that received the method invocation. For now, we will for class-based languages present residual programs as a list of specialized methods of the form:

```
T C.m( parameters ) { body }
```

Here, `m` is a specialized method that belongs to the class `C`, with return type `T`, formal parameters "`parameters`," and body "`body`." Section 5.3 properly investigates the representation of specialized programs in the case of class-based languages, and Chapter 8 addresses object-based languages.

5.2.5 Example: power

The power example shown in Figure 5.1 can be specialized in a number of different ways, akin to what was done for the Scheme and C versions of power. When the exponent value is known, specialization unfolds the recursive calls to `loop`, as shown for an exponent value of 3 with the `Power.raise_3` method in Figure 5.4a. When the operator and neutral value are known in addition to the exponent (as in the graphically illustrated scenario of Figure 5.3), specialization eliminates all overheads due to object-oriented mechanisms. The virtual dispatches to the binary operator are unfolded, and the neutral value is inserted directly into the specialized code, as shown with the `Power.raise_3_Mult_1` method in Figure 5.4a. Optimizing this specialized method yields the simplified version `Power.raise_3_Mult_1_opt`.

As an example of speculative evaluation, we can assume that the exponent and base value are known, but that the operator and neutral value are unknown.

The result of specialization is shown in Figure 5.4b. Specialized versions of each of the binary operators is generated for use in each application of the `eval` method, since it here was applied with a known first argument (the base value). A virtual dispatch that invokes one of these specialized methods has been residualized for each recursive invocation of the method `loop`.

5.3 Aspects and Specialized Program Appearance

Aspect-oriented programming is concerned with logical units of functionality that cut across the program structure; it allows these logical units to be separated from other parts of the program, encapsulated into aspects. Partial evaluation of an object-oriented program produces a set of specialized methods, that in a class-based language must be inserted into the existing program structure. These methods are separate from the main program but have dependencies that cross-cut the program structure, and can be viewed as an aspect of the program.

This section proposes aspects as a means for expressing specialized programs and preserving their modularity, and gives perspectives on partial evaluation based on aspect-oriented programming. We concentrate on class-based languages; object-based languages will be discussed in Chapter 8. First, we consider modularity issues in specializing programs using partial evaluation. Then, we give a brief overview of aspect-oriented programming and discuss its relation to partial evaluation. Last, we present an aspect-oriented, modular approach to expressing the result of specializing an object-oriented program.

5.3.1 Modularity and partial evaluation

There are several issues to take into account when specializing programs written in a modular language. In this section we discuss modularity and maintenance of specialized code, introduce encapsulation of specialized code, and present existing efforts to reconcile partial evaluation and modular languages.

Modularity and maintenance

A specialized method may need to reference local attributes defined in the receiver object for which it was specialized. In a class-based language, a specialized method can either be added to the class of the object, or be defined independently of the class. In the latter case, the program must somehow be restructured to permit access to the attributes of the object without breaking encapsulation.

To make partial evaluation work for realistic programs, modular specialization (i.e., specialization of a program slice and reinsertion of specialized code into the generic code) is needed [41] (see also Chapter 9). Hence, the class structure of the program part being specialized can only be changed if it remains compatible with the rest of the program. To avoid the complications caused by modifying the class structure of the program, we choose to add specialized methods to classes. However, if the specializer were to directly add specialized

```
class SafeWithKey {
 private int key;
 private Object content;
 SafeWithKey( int k, Object c ) {
  key = k; content = c;
 }
 Object get(int k) {
  return (k!=key)?null:content;
 }
}
class Client {
 ...
 void access( SafeWithKey s, int k ) {
   Object x = s.get(k);
   ...
 }
 void entrypoint( ... ) {
   ...
 }
}
```

(a) Generic program

```
class SafeWithKey {
 ... (fields and generic methods) ...
 Object get_spec() {
  return content;
 }
}
class Client {
 ... (generic methods) ...
 void access_spec( SafeWithKey s ) {
  Object x = s.get_spec();
  ...
 }
 void entrypoint_spec( ... ) {
  ...
 }
}
```

(b) Specialization for a (correct) key

Figure 5.5: Specialization producing unsafe methods.

methods to the classes of the program, it would mix generic program code and specialized program code; such a mix would obfuscate the functionality of the program and complicate program maintenance. Alternatively, specialization could create a separate copy of the program with classes containing a mix of generic and specialized methods. However, this approach could easily lead to inconsistencies, since the unspecialized parts of the program may be modified after specialization as part of the continued development of the program. We need a modular representation of the specialized program part.

Preserving encapsulation

Encapsulation combined with access modifiers facilitates reasoning about properties local to an object, since methods and fields private to a specific object must be manipulated from within that object. Adding specialized methods to a class may inadvertently break encapsulation invariants, since checks or computations that ensure an invariant may be specialized away. As an example, consider the program shown in Figure 5.5a. Here, an object of class `Client` uses an integer key to retrieve some object stored in a `SafeWithKey` object; the contents of the `SafeWithKey` object only can be accessed when a correct key is supplied. Specialization of this program with direct reintegration of methods into classes would give rise to the program shown in Figure 5.5b. Here, specialization for a static key value has produced the specialized method `get_spec` that no longer performs a key check. By construction, the method `get_spec` is only called from a context where a correct key was supplied. Nevertheless, an improper subsequent modification of the program could give rise to a call to `get_spec` from a context where the correct key is not available. With the exception of the specialization entry point, specialized methods should only be called from other specialized methods, to ensure that they are used in a correct context. The standard object-oriented encapsulation mechanisms cannot be used to express that certain methods in one class may only be called from certain other methods in other classes. We need a mechanism that expresses encapsulation across the classes of the program.

Partial evaluation and modularity

The problems of adding specialized methods to a class in a modular way and preserving encapsulation appears to be a general problem in partial evaluation for modular languages: how to integrate specialized methods (functions, procedures) back into the generic modules that constitute the program?

To this date, very few partial evaluators take modularity in the source language into account. Partial evaluators for SML are usually defined for a language subset that excludes the module system [17, 105], and the only partial evaluator for a modular imperative language (Modula-2) is not documented in an easily accessible form (it is described in Russian, and mentioned in a paper by Bulyonkov and Kochetov [27]). Integration of specialized code is straightforward in languages without modules. As an example, consider the partial evaluators Schism and Tempo, with which the author is familiar (Schism basically treats a subset of the functional language Scheme, Tempo treats the entire C language). Schism produces a set of specialized functions to be loaded into the top-level environment, and Tempo produces a self-contained C file that can be compiled into an object file and then linked with the generic program. The lack of modules in these languages makes introduction of specialization program code into the generic program trivial.

Dussart, Heldal and Hughes study partial evaluation for a small, modular functional language. This approach consists of analyzing each program module independently, and then specializing them together to form a complete specialized program [53]; this approach is referred to as *module-sensitive program specialization*. To overcome visibility issues, the module language declarations that control visibility are changed in the specialized program, and specialized methods may be moved from one module to another in the presence of higher-order functions. Neither of these solutions are acceptable in the context of object-oriented languages. First, there are no standard object-oriented declarations to control visibility of attributes between two distinct classes. Second, in a language with access modifiers, private members can only be accessed from within the object itself. A specialized method may need to access receiver object fields that contain dynamic data, so specialized methods cannot be moved freely between objects. Furthermore, this specialization approach is global, i.e., all the modules of a program are specialized, and integration with existing generic modules is not takes into account. We need a solution that respects the existing module structure.

5.3.2 Aspect-oriented programming and partial evaluation

Traditional program structuring mechanisms, such as procedures and classes, often fall short when expressing program functionalities that have a global extent. For example, the implementation of synchronization or distribution features often ends up dispersed across the entire program rather than localized in a single place. Such distribution of the implementation of a single functionality throughout the program breaks modularity and obfuscates program structure, which complicates program maintenance and development. Kiczales et al. classify logical features that *cross-cut* the program structure as *aspects*, and present *aspect-oriented programming* as a means of expressing the implementation of such features in a modular way [91]. Here, aspects form individual modules that together with program parts written in standard language constructs form a complete program. To compile an aspect-oriented language built over, e.g., an object-oriented language into this object-oriented language, a dedicated compiler called a *weaver* combines aspects and object-oriented program parts to form a standard object-oriented program that can be compiled and executed.

Partial evaluation transforms an object-oriented program by adding new methods to the classes of the program; it is these methods that implement a specialized version of the program behavior. Each newly added method is specialized to a specific context, and is only used by other specialized methods (apart from the specialization entry point, which is designated by the user). Methods are added across the classes of the program, their caller-callee dependencies *cross-cutting* the class structure of the program. We can view the methods that are added to the program as an aspect of the program that defines a specific specialized behavior. The aspect consists of the specialized methods combined with an indication of what class to add them to. To form a complete specialized program, this aspect can combined with the generic program by a trivial weaver program that adds the specialized methods to appropriate classes to form a specialized, object-oriented program.

5.3.3 Specialization as an aspect

We propose to place specialized methods in an aspect; specializing an object-oriented program yields an aspect-oriented program, which is passed to a weaver to produce a standard object-oriented program. With this approach, we retain a clear distinction between generic program code and specialized program code. Code clarity and readability are improved since the program-wide effect of specialization is collected in a single aspect that represents the specialized code. Encapsulation is improved since only the specialization entry point needs to be visible outside the aspect; the specialized functions are encapsulated within the aspect. Modularity is improved since the decision of whether to use a specific specialized version of the program is handled by selecting what aspects to weave into the program.

Concretely, we choose a simplified version of the aspect-oriented language AspectJ [168] as an aspect language for expressing specialized program code. AspectJ exists and is a well-known aspect-oriented extension to Java; it has a convenient syntax, and it includes most of the features that we require. The current version of AspectJ has no means of explicitly controlling encapsulation within the aspect, but this is apparently planned for a future version of the

```
ASPECT         ::=  aspect NAME {
                     (INTRODUCTION)+
                    }
INTRODUCTION   ::=  MODIFIER introduction CLASS-NAME {
                     (JAVA-METHOD)+
                    }
MODIFIER       ::=   private | public
NAME           ::=   ...(standard Java name)...
CLASS-NAME     ::=   ...(standard Java class name)...
JAVA-METHOD    ::=   ...(standard Java method declaration)...
```

Figure 5.6: Syntax of an aspect language for expressing specialized programs.

```
aspect NoKey {
  private introduction SafeWithKey {
    Object get_spec() {
      return content;
    }
  }
  private introduction Client {
    void access_spec( SafeWithKey s ) {
      Object x = s.get_spec();
      ...
    }
  }
  public introduction Client {
    void entrypoint_spec( ... ) {
      ...
    }
  }
}
```

Figure 5.7: Specialized code of Figure 5.5 as an aspect.

language [90]. Indeed, the use of aspect access modifiers to control visibility of specialized methods introduced into a class was originally proposed by Kiczales [89]. The syntax of our aspect language is shown in Figure 5.6. An aspect consists of a collection of `introduction` blocks. An `introduction` block applies to a given class, and introduces the methods listed in the body of the block into the class. To express encapsulation, we have taken the straightforward solution of adding an access modifier to each `introduction` block; a method introduced in a private block is visible only to other methods defined in the same aspect, and a method introduced in a public block is globally visible.

As an example of using our aspect-oriented language to express a specialized program, consider the `SafeWithKey` example of Figure 5.5. Here, the method `get` of the class `SafeWithKey` was specialized to no longer check the `key` argument. Using aspects, we can express the specialized program as shown in Figure 5.7.

The aspect `NoKey` introduces the method `get_spec` into the class `SafeWithKey` and the method `access_spec` into the `Client` class. Both methods are declared as private to the aspect; this way, they can only be called by methods declared in the aspect. The specialization entry point is introduced into the `Client` class as well, but publicly, so that it can be called by user-defined code from outside the aspect.

Weaving the specialized program aspect with the generic program produces a specialized object-oriented program. All modularity and encapsulation benefits due to the aspect-oriented approach are lost in the resulting program, but standard tools can be used to compile and execute the program. The loss of modularity and encapsulation is problematic but inevitable; the problem can be minimized by always weaving before compilation of the specialized program. Indeed, the AspectJ weaver automatically calls the Java compiler, which makes the entire process transparent to the user.

5.4 Specialization Examples

To further illustrate how specialization transforms a program, in this section we revisit the power example, and investigate specialization of an interpreter for arithmetic expressions.

5.4.1 Power revisited

The specialized versions of the power program shown in Figure 5.4 are shown in Figure 5.8 using our aspect-oriented notation. The aspect `Power_exp_op_neutral` is the result of specializing the power program to a given exponent, operator and neutral value. The power computation has been simplified to its most basic form. The aspect `Power_exp_base` is the result of specializing the power program to a given exponent and base value; specialized `eval` methods have been generated for each application of the unknown operator onto the known base value.

5.4.2 Arithmetic expression interpreter

Partial evaluation works well when used to eliminate layers of interpretation. As a special case, when used to specialize an interpreter to a given input program, partial evaluation can unfold all interpreter computations that depend on the input program. In effect, partial evaluation of an interpreter implements compilation from the interpreter input language to the interpreter implementation language (this is called the First Futamura projection [63]). We can apply partial evaluation to the simple interpreter for arithmetic expressions shown in Figure 5.9. Such an interpreter could for example be used in a program to compute the value of an expression supplied by the user. All interpreter classes inherit from the `Exp` class and have an `eval` method parameterized by an environment; this method computes the value of the expression represented by the object, given the values provided by the environment. The classes `Exp`,

```
aspect Power_exp_op_neutral {
 public introduction Power {
  int raise_3_Mult_1(int base) {
   return base*base*base;
  }
 }
}
```

(a) Standard specialization of power

```
aspect Power_exp_base {

    public introduction Power {           private introduction Add {
     int raise_3_1() {                     int eval_2( int y ) {
      return this.op.eval_2(                return 2+y;
       this.op.eval_2(                     }
        this.op.eval_2(                   }
         this.neutral)));
     }                                    private introduction Mult {
    }                                      int eval_2( int y ) {
                                            return 2*x;
    private introduction Binary {          }
     int eval_2( int y ) {                }
      return this.eval_2( y );
     }
    }

}
```

(b) Speculative specialization of power

Figure 5.8: Specialized power in different versions.

`Binary` and `Unary` act as abstract classes (as usual, their `eval` method is defined directly in terms of itself).

We specialize the interpreter to the expression $(x_0 + (-x_1)) \times (x_0 + (-x_1))$ which has the effect of compiling it into equivalent Java code. The Java representation of the expression is shown in Figure 5.10a, and the program that results from specializing the interpreter to this expression is shown in Figure 5.10b, after inlining optimizations. All virtual dispatches have been removed by specialization, and only the basic operations manipulating dynamic data (values stored in the environment) have been residualized.

5.5 Summary

In object-oriented languages, a program is regarded as a collection of objects that interact by message passing. Partial evaluation specializes an object-oriented program by eliminating those interactions that are fixed given the static input and simplifying those interactions that depend partly on static input. It thereby reduces the number of object interactions performed during a run of the program and reduces their individual computational cost. The result of specialization is an aspect that enhances the objects that constitute the program with the specialized behavior.

```
class Exp {
 int eval(int[] env) {
  return this.eval(env);
 }
}
class Constant extends Exp {
 int c;
 Constant(int i) { c=i; }
 int eval(int[] env) {
  return c;
 }
}
class Var extends Exp {
 int v;
 Var(int i) { v=i; }
 int eval(int[] env) {
  return env[v];
 }
}
class Binary extends Exp {
 Exp left, right;
 Binary(Exp l, Exp r) {
  left=l; right=r;
 }
 Exp getLeft() { return left; }
 Exp getRight() { return right; }
}
```

```
class Add extends Binary {
 Add(Exp l, Exp r) { super(l,r); }
 int eval(int[] env) {
  return getLeft().eval(env)
       + getRight().eval(env);
 }
}
class Mul extends Binary {
 Mul(Exp l, Exp r) { super(l,r); }
 int eval(int[] env) {
  return getLeft().eval(env)
       * getRight().eval(env);
 }
}
class Unary extends Exp {
 Exp e;
 Unary(Exp x) { e=x; }
 Exp getExp() { return e; }
}
class Neg extends Unary {
 Neg(Exp x) { super(x); }
 int eval(int[] env) {
  return -getExp().eval(env);
 }
}
```

Figure 5.9: Arithmetic expression interpreter.

```
new Mul(
 new Add(
  new Var(0),
  new Neg(
   new Var(1))),
 new Add(
  new Var(0),
  new Neg(
   new Var(1))))
```

(a) Input to specializer

```
aspect SquareOfDiffsOptimized {
 public introduction Mul {
  int eval_spec(int[] env) {
   return (env[0]+(-env[1]))
        * (env[0]+(-env[1]));
  }
 }
}
```

(b) Specialized program

Figure 5.10: Specialization of arithmetic expression interpreter.

Concretely, partial evaluation propagates static values from method arguments or object fields throughout the program, and reduces any decisions or computations that are based solely on these values. A virtual dispatch is a decision over the receiver type; it is reduced when the receiver is static, and it is specialized speculatively by specializing each potential callee when the receiver is dynamic. Specialization of an object-oriented program generates a set of specialized methods encapsulated in an aspect; the aspect preserves modularity and respects encapsulation in the specialized program. Weaving compiles the aspect and the generic program into a specialized object-oriented program.

We have in this chapter shown how partial evaluation specializes an object-oriented program; it removes overheads due to generic use of object-oriented abstractions. Partial evaluation for object-oriented languages provides similar functionality to partial evaluation for functional, logical and imperative programs, namely an automatic mapping from a generic program into a specific implementation.

Chapter 6

Declarative Specialization

6.1 Introduction

Partial evaluation can specialize an object-oriented program to a specific pattern of usage; it automates tasks that would otherwise have been performed manually during software development. But for partial evaluation to be a useful object-oriented software engineering tool, its use must be integrated with the software development process. Concretely, an approach is needed that integrates well with object-oriented programming, and that offers easy access to the standard features of a partial evaluator.

Traditionally, relatively little effort has been devoted to offering a simple, high-level interface to partial evaluators; the development of analysis and transformation features is usually given a higher priority. In the case of functional languages, the need for a high-level interface is limited; only the basic binding times and actual values of the arguments to the entry point need to be specified. For imperative languages, more information must be specified, and the need for a high-level interface is more accentuated; information about global variables and alias relations between locations must, for example, be specified. In both cases, the user must, in a realistic setting, also provide information about the behavior of the code surrounding the program slice being specialized. As an example, consider the Tempo partial evaluator [41]. Here, a set of obscure configuration files in SML and C define parameters to the partial evaluator and describe the specialization context. The more complex a partial evaluator and the language that it treats, the greater the need for a well-designed partial evaluator interface.

In recognition of the need for a uniform approach integrated with the target language, Volanschi et al. present a declarative framework for controlling partial evaluation [162]. In this framework, named *specialization classes*, declarations referred to as specialization classes control partial evaluation of Java programs. A *specialization class* declares a set of invariants for which the methods of a class in the program should be specialized. Specialization classes allow the user to directly declare the invariants for which to specialize, as opposed to programming the appropriate invariants in an operational configuration language (such as that of Tempo). The specialization class framework automatically re-integrates specialized code into the program; it augments the program with guards that automatically enables the specialized code to be used when the specialization invariants are satisfied.

Specialization classes were conceived before partial evaluation for object-oriented languages was defined; it turns out that they do not offer sufficient power of expression to adequately control specialization of object-oriented programs that make use of object-oriented abstractions. Furthermore, specialization classes have been presented as a means of specializing complete programs; their application during program development has not been taken into account in their design. We have extended the specialization class framework to match the needs of object-oriented partial evaluation, and we give guidelines for using specialization classes to add specialization capabilities to a program during its development.

This chapter describes specialization classes, a declarative framework for controlling object-oriented partial evaluation. We present the existing work on specialization classes as well as our own extensions to this work. We demonstrate that specialization classes should be seen as a form of aspect-oriented programming, and show how to use specialization classes in the software engineering process. The extended specialization class framework described in this chapter has been implemented in conjunction with the JSpec partial evaluator for Java presented in Chapter 10.

Chapter overview: First, Section 6.2 presents the original specialization class framework. Then, Section 6.3 presents our extensions to this framework, including complete examples of specialization class declarations. Afterward, Section 6.4 compares specialization classes with aspect-oriented programming, and Section 6.5 describes how to use specialization classes during software development. Last, Section 6.6 presents a summary.

Notes: Specialization classes work straightforwardly with class-based object-oriented languages, but it is not obvious how to adapt the specialization class framework for working with an object-based language. We consider the development of such a framework as future work. The syntax and semantics of the specialization class extensions presented in Section 6.3 were developed in collaboration with Helle Markmann Andersen as part of her DEA project, which was supervised by the author. She also developed an initial prototype of the specialization class compiler, for use with the Java partial evaluator presented in Chapter 10.

6.2 Specialization Classes

Specialization classes were defined by Volanschi et al. as a framework for specializing object-oriented programs; the primary contribution of specialization classes is to offer a declarative approach to controlling partial evaluation. Automatic re-integration of specialized code is known from other partial evaluation systems [11, 62], but specialization classes are the first to offer this feature independently of the language compiler. Specialization classes rely on certain Java features (access modifiers and interfaces) for implementation simplicity, but are in principle independent of the language.

This section first presents the basic features of specialization classes, describes how specialization classes can be combined to offer more advanced features, and gives an example of using specialization classes to specialize a program.

6.2.1 Specialization class basics

A specialization class declaration indicates a set of program components to specialize, defines the context for which these components should be specialized, and defines any options needed to configure the partial evaluator. The program components that can be specialized are methods (these methods are referred to as the *target methods*); a specialization class can name multiple target methods that should be specialized together. The specialization context is defined by declaring invariants over the receiver object of the target methods. Options defined in the specialization class declaration can be passed to the partial evaluator, which allows details in the control of the partial evaluator to be integrated into the specialization class.

Based on the declaration of what components to specialize, a dedicated compiler extracts the target methods from the program and uses a partial evaluator to specialize them to the context defined by the invariants. This compiler, referred to as the *specialization class compiler*, modifies the original program by replacing the target methods with special guarded methods. These methods correspond to wrappers; they call the specialized methods when the invariants are satisfied, and call the generic methods otherwise. (The guarded methods are akin to predicate dispatching [31], but they do not rely on any form of multi-dispatching in the language; the specialization class compiler generates all the necessary dispatching code.)

6.2.2 Combining specialization classes

A specialization class can either apply directly to some class in the program, and declare specialization of methods found in this class, or extend some other specialization class. The latter case is called a *derived specialization class*; it represents a refinement of the specialization class that it extends, mimicking object-oriented inheritance. A derived specialization class can declare additional methods to specialize and define more precise invariants.

Specialization class invariants over base type values are specified straightforwardly by providing a concrete value. Invariants over reference types are more complex, since information about a complete object is declared. A specialization class specifies invariants over object values in terms of delegate specialization classes, mimicking object-oriented delegation. Specifically, delegate specialization classes are used to specify invariants over object references; an invariant where an object reference is qualified by a delegate specialization class

```
specclass Cube specializes Power {
  exp == 3;
  SpecMult op;
  neutral == 0;
  int raise( int base );
}
specclass SpecMult specializes Mult {}
```

Figure 6.1: Specialization class for power example.

is satisfied when the referenced object satisfies the invariants of the delegate specialization class. The specialization class compiler implements a communication protocol between objects that permits testing whether some object satisfies the invariants of a given specialization class.

6.2.3 Specialization class example

In the power example from the previous chapter, the `raise` method was specialized for a specific exponent, operator and neutral value (see Figures 5.3 and 5.8). Using specialization classes, this scenario can be declared as shown in Figure 6.1. Here, a specialization class named `Cube` declares specialization of the method `raise` of the class `Power`. The fields `exp` and `neutral` are given the concrete values for which they should be specialized, and the `op` field is qualified by the delegate specialization class `SpecMult`. The specialization class `SpecMult` does not declare any invariants, but implicitly serves to declare that the type of the object is `Mult`. However, with the existing definition of specialization classes, it is unclear whether the invariant over the `op` field can be used to deduce the static information that `op` has type `Mult`; the specialization class framework is not designed to express such information.

6.3 Extended Specialization Classes

We have extended the specialization class framework based on our investigation of partial evaluation for object-oriented languages. Like the original specialization class framework, the extended specialization class framework is intended to work with Java as a host language, and uses a Java-like syntax.

In this section, we first present our extensions to the original specialization class framework, and then describe the syntax and semantics of the extended specialization class declarations. Afterward, we give complete examples of specialization class declarations, and outline the current state of the specialization class compiler implementation. As an example throughout this section, we will use the power and arithmetic interpreter examples of the previous section (see Figures 5.3, 5.8, 5.9 and 5.10).

6.3.1 Extensions

The original specialization class framework only permits abstract invariants over base type values. The extended specialization class framework allows abstract invariants to be expressed over any kind of value, and generalizes the possibilities for specifying invariants using delegate specialization classes. Abstract invariants are essential for run-time specialization, but are in general useful when the user does not want to specify the exact data for which specialization should take place; this is how partial evaluators are often used. In the arithmetic interpreter example, binding-time analysis should be performed for an abstract expression, and the interpreter can then be specialized for any concrete expression.

Partial evaluation of an object-oriented program simplifies the interaction that takes place between the objects that constitute the program. The object interaction is in part controlled by the way objects are composed together. As a consequence, we need to express invariants over object structures using specialization classes. However, the original specialization class framework does not permit recursive invariants over object structures, and does not define a clear semantics for how specialization classes interact with object-oriented inheritance. Conceivably, a specialization class can either apply to instances of a class and all of its subclasses, or only to instances of the specific class mentioned in the specialization class declaration. It is only in the latter case that precise information regarding object composition can be extracted from specialization class declarations. In the extended specialization class framework, a specialization class applies only to the program classes listed in its declaration. Furthermore, the extended specialization class framework does not impose restrictions on the shape of the object structure that can be specified using specialization class invariants. For clarity, the extended specialization classes have an unambiguous syntax and semantics that allow the specification of precise invariants over object references. In the power and arithmetic interpreter examples, it is essential that precise information about object composition can be expressed, to allow virtual dispatches to be reduced.

The original specialization class framework only permits invariants over the self argument to a method, not over other method arguments. In practice, it is essential to permit specialization over all arguments to a method. Such invariants are supported by the extended specialization class framework, similarly to how specialization over the self object is supported by the original specialization class framework. In the power example, invariants need to be expressed over the parameters of the `raise` method.

6.3.2 Syntax and semantics

The extended specialization class syntax is shown in Figure 6.2; we refer to Volanschi's work [161, 162] for the original specialization class syntax. The syntax presented here does not include options to the partial evaluator, although it has been designed with such options in mind. We first describe the syntax and semantics of class-level declarations; then, we describe field-level declarations

```
SPEC_CLASS      ::=  specclass SC_NAME PARENT {
                       (PREDICATE; | METHOD)*
                     }
PARENT          ::=  specializes CLASS_NAME (, CLASS_NAME)*
                 |   extends SC_NAME
PREDICATE       ::=  VARIABLE_DECL == VALUE_DECL
                 |   VARIABLE_DECL :  TYPE_DECL
                 |   VARIABLE_DECL :< TYPE_DECL
                 |   this :  TYPE_DECL
                 |   this :< TYPE_DECL
METHOD          ::=  @specialize :  METHOD_PROTOTYPE PARAM_DECL
                 |   @specialize :< METHOD_PROTOTYPE PARAM_DECL
VARIABLE_DECL   ::=  FIELD_NAME
                 |   PARAMETER
PARAM_DECL      ::=  where PREDICATE;
                 |   {(where PREDICATE;)+}
                 |   ;
VALUE_DECL      ::=  VALUE
                 |   !
                 |   ...
TYPE_DECL       ::=  ...(array declaration)...
                 |   TYPE (| TYPE)*
TYPE            ::=  SC_NAME
                 |   CLASS_NAME
                 |   TYPE_NAME
SC_NAME         ::=  IDENTIFIER
```

Figure 6.2: Grammar of extended specialization classes.

and method-level declarations; last, we describe how a set of specialization class declarations specialize a program.

Class-level declaration

A specialization class either specializes a number of Java classes or extends some other specialization class. A specialization class that specializes a number of Java classes (referred to as the *root classes*) can declare invariants over the state of any object that is an instance of one of these classes, and can declare specialization of methods defined in these classes. A specialization class declaration with more than one root class is equivalent to a duplicated declaration for each individual root class. In the arithmetic interpreter example, we can declare invariants over the classes `Add` and `Mul` as follows:

```
specclass SpecBinaryOperator specializes Add, Mul { ... }
```

which is equivalent to two individual declarations. However, any reference to the specialization class `SpecBinaryOperator` includes both classes. Multiple root classes can be regarded as syntactic sugar, and we will for simplicity describe specialization classes using only a single root class.

There is a dual purpose to extending a specialization class by some other specialization class: the declarations of the specialization class are reused, and there is an explicit ordering between the two specialization classes. Consider the following declarations:

```
specclass X specializes A { ... }
specclass Y extends X { ... }
```

Here, any invariants or methods declared in `X` are reused in `Y`. In addition, when the invariants of both `X` and `Y` are satisfied, the explicit ordering makes it trivial to determine that `Y` is more specific than `X`, and therefore is assumed to provide a better specialization. (The extension of one specialization class by another was exploited by Volanschi et al. to define gradual invariants over operating system components [162].) The root class of a specialization class that extends some other specialization class is the root class of the class that it extends.

Field declarations

A specialization class can declare invariants over the fields of its root class; an object that satisfies all the invariants of a specialization class is said to be in the state of this specialization class. An invariant over a field can specify equality to a base-type value using "`==`", or can specify a type relation using the type operator "`:`". In the power example, the invariant that the exponent has the value `3` is written "`exp == 3;`", and the invariant that the operator has exactly the type `Mult` is written "`op: Mult;`". Invariants that give an abstract description of the state of an object are written using a similar syntax. A known base type value can be indicated using "`!`", e.g., the invariant "`exp == !;`" specifies that the exponent is known. A known type can be qualified using the sub-type operator "`:<`", e.g., the invariant "`op:< Binary;`" specifies that the operator is known and is of the class `Binary` or any subclass of `Binary`. Fields not covered by invariants are considered to be unknown and thus dynamic. Multiple types can be joined together using the "`|`" operator, to indicate that a field has either one or the other type, e.g., the invariant "`op: Add | Mult`" indicates that the operator is known and has type `Add` or `Mult`.

In addition to invariants over the fields of an object, there is a default invariant that the type of an object is the type of the root class. By specifying a type invariant using the keyword `this`, the default invariant can be overridden. For example, in the arithmetic expression interpreter example, the invariant "`this:< Exp`" could be used in a specialization class to indicate that the self object can be any subclass of `Exp`.

We say that an object has the type of a specialization class if it is in the state defined by the specialization class. Using this convention, the type invariant operators "`:`" and "`:<`" can be used to specify delegate specialization classes. In the arithmetic interpreter example, the specialization declaration

```
specclass SpecNeg specializes Neg {
  e: SpecAdd | SpecMult;
}
```

can be used to indicate that the object field `e` has the type of the delegate specialization classes `SpecAdd` and `SpecMult` (which must also be defined).

To concisely specify that several fields are covered by the same invariant, multiple fields can be listed on the left-hand side of a type operator. Furthermore, invariants can be expressed over array types and the contents of arrays (not shown in grammar).

Method declarations

A method to specialize is indicated using the keyword `@specialize` with a type operator. Here, the "`:`" operator indicates specialization of a method that belongs to the root class, and the "`:<`" operator indicates virtual method specialization (see next paragraph).[1] Invariants over the formal parameters of a method are declared using the keyword `where`; invariants over formal parameters are written in the same syntax as invariants over object fields. A specialized method is used when all invariants over the self object and its formal parameters are satisfied for a given invocation of the method. In the power example, specialization for a base value of `2` can be done using the declaration

```
@specialize: int raise( int base ) where base==2;
```

A set of specialization classes for specializing a program can declare multiple methods to specialize; these are in principle specialized independently, although an implementation is free to share specialized code between specialized methods.

A "`@specialize`" declaration with a subclass operator "`:<`" indicates virtual method specialization. Although any kind of method can be specialized with the standard "`@specialize :`" declaration, it can be useful to let the choice of what method to specialize depend on the concrete values for which specialization is done. Virtual method specialization is useful in conjunction with an invariant that uses the keyword `this` to indicate that the self object is an instance of one of a number of different classes. In this case, virtual method specialization allows any method matched by the `@specialize` declaration and declared in one of these classes to be the specialization entry point. It is only when a concrete value is supplied that a specialization entry point is chosen. In the arithmetic interpreter example, the entry point method `compute` is redefined by each subclass of the class `Exp`; the declaration

```
@specialize<: int compute( int[] env );
```

allows the `compute` method of the object of the root of the expression for which specialization should be done to be the specialization entry point.

Specialization

Specialization of a program to a collection of specialization classes is done as follows. Each `@specialize` method declaration is considered a specialization entry point. Specialization of an entry point method is done in two steps; the method is analyzed in the abstract context defined by the invariants over the method parameters and the self object; then the method is specialized to the concrete data supplied by the user. After specialization, the generated method is re-integrated into the program under a different name (along with any specialized methods that it uses, as shown in the previous chapter), and the original method is modified by inserting guards. The guards ensure that

[1]We envision adding more keywords like `@specialize` that indicate properties about methods, for example whether a method is visible during analysis, or whether it can be called during specialization.

```
specclass Cube specializes Power {
  exp == 3;
  op : Mult;
  neutral == 0;
  @specialize : int raise( int base );
}
```

Figure 6.3: Specialization class for power example (revised).

when the runtime state matches the concrete data for which a method was specialized, this method will be used.

The context in which the specialization entry point should be analyzed can be derived directly from the specialization class declarations. A specialization class represents one or more object instances of the corresponding root Java class, that all have binding times matching the abstract description given in the specialization class. A type invariant over a field or a parameter indicates that the object referenced through the field or parameter should conform to this type. When the subclass operator "`:<`" is used, the object referenced may be of any of the listed types or any of their subtypes. An invariant specifying the type of `this` allows the self object to have any one of the types listed (or their subclasses, when using the subclass operator).

Several specialization classes can be applied to the same Java class. When a given object is in the state of two or more specialization classes, the specialized methods associated with the more specific specialization class should be used. If the specialization classes are ordered by the `extends` relation, it is guaranteed that the most specific specialization class will be used. Otherwise, there might not be a most specific specialization class, and the choice is in principle non-deterministic. When multiple specialized methods are generated from the same specialization class, a similar problem arises. The invariants that guard two different specialized methods may both be true in a given program state; we take the simple solution of letting the choice between these methods be non-deterministic.

6.3.3 Examples

The complete specialization class of the power example is shown in Figure 6.3. The `exp` and `neutral` fields are given concrete values, the `op` field is declared to have type `Mult`, and the `raise` method is declared as being the specialization entry-point.

The complete specialization class of the arithmetic expression interpreter example is shown in Figure 6.4. The specialization class `SExp` defines the specialization entry point; the self object is declared to match one of a set of specialization classes. Each of these delegate specialization classes is declared as one of the object types that can appear in the expression; invariants over

```
specclass SExp specializes Exp {
  this : SConstant | SVar | SBin | SUna;
  @specialize :< int compute( int[] env );
}
specclass SConstant specializes Constant {
  c == !;
}
specclass SVar specializes Var {
  v == !;
}
specclass SBin specializes Add, Mul {
  left, right : SConstant | SVar | SBin | SUna;
}
specclass SUna specializes Neg {
  e : SConstant, SVar | SBin | SUna;
}
```

Figure 6.4: Specialization classes for arithmetic expression interpreter example.

fields are used where the object contents are known. The `SConstant` and `SVar` specialization classes declare their fields (a constant value and a variable number, respectively) to be static. The `SBin` and `SUna` specialization classes declare that the children of the node represented by the object are known as well. The specialization entry point is declared virtual, to make the choice of specialization entry point depend on the concrete data that is supplied.

6.3.4 Implementation

The specialization class compiler for the extended specialization class framework has been integrated with the JSpec partial evaluator for Java described in Chapter 10. However, the current implementation only allows a single specialization entry point, and it only supports automatic generation of the analysis context; we return to this limitation below.

The analysis context can be generated straightforwardly from the specialization class declarations. An object is allocated for each specialization class, and its fields are initialized according to the invariants. A base type field is initialized using the JSpec interface for creating static or dynamic base type values. A field qualified by a specialization class gives rise to an assignment of this field to the object that represents the specialization class. A field qualified by a Java type is simply assigned a newly created object. Type invariants that are joined using the "`|`" operator give rise to a conditional that either performs the assignment corresponding to the one type invariant or the other type invariant. This conditional has a static test, which causes the binding-time analysis to merge the result of each branch and consider the result to be static.

The current implementation of the specialization class compiler only permits a single specialization entry point and does not automatically generate guards. An initial implementation of multiple specialization entry point support is planned by invoking the partial evaluator once for each entry point. This approach may however result in duplicate specialized methods, since specialization of different entry points can produce intersecting sets of specialized methods.

Although not implemented, the generation of guards has been taken into account in the design of the extended specialization class framework. Guards should be generated based on the concrete data supplied to the specializer, not the abstract description declared using specialization classes. To generate guards, the specializer must determine the actual values stored in the concrete data. Based on the abstract description in the specialization classes, this data can be determined either by using the Java reflection mechanism, or by instrumenting the classes of the program so that they can report the values of their fields. The guards that are generated can be introduced into the program using the aspect-oriented approach described in the previous chapter. In particular, AspectJ cross-cuts that are executed "around" the body of a method permit code to be inserted that guards the execution of a method body [167]. This construction allows a guard to be inserted around the unspecialized method body that selects between the unspecialized method body and any specialized method bodies.

6.4 Specialization Classes as Aspects

As demonstrated in this chapter, object-oriented partial evaluation can be controlled by specialization class declarations separate from the program. In fact, a set of specialization classes define a specific specialized aspect of the program behavior. Hence, specialization classes can be seen as an aspect-oriented extension to an object-oriented language: specialization classes declare knowledge that can be used to optimize the interaction that takes places between objects, and divide the computations of the program into static and dynamic. As for the problems that may arise when multiple, conflicting specialization classes apply to the same program class, these are similar to the problems that arise when composing aspects that apply to the same program part. We believe that aspect-oriented programming may provide inspiration as to how these conflicts can be resolved, but we consider this investigation to be future work.

If the aspect resulting from specialization is seen as the output of the partial evaluator, the partial evaluator can be seen as transforming a declarative aspect (the specialization class declarations) into an operational aspect (the AspectJ-like language describing what methods to add to the classes of the program). Conversely, if the aspect resulting from specialization is considered to be woven directly into the source program as part of the specialization process, the partial evaluator can be seen as a weaver: it combines the program and the specialization classes to produce a program, optimized (specialized) according to the information contained in the specialization class declarations.

6.5 Software Development and Specialization Classes

For partial evaluation to be a useful software engineering tool, specialization of a program must be integrated into the program as it is being developed. Using specialization classes, various specialization scenarios adapting the program to a specific usage context can be developed and extended to match program development needs. In practical use, the declarative nature of specialization classes greatly simplifies the application of partial evaluation; the focus on what to specialize in a program as opposed to how to specialize it makes the specialization specification almost self-documenting.

During analysis and design of an object-oriented program, related concepts may be decomposed into distinct objects. When there are program usage scenarios where the relation between these objects is fixed, a specialization class can be introduced by the programmer to declare the nature of this relation. Similarly, when an object is parameterized by a number of attributes that are invariant in a given usage scenario, specialization classes can be introduced to declare this fact. The knowledge of when the relation between two objects may be fixed or when a parameter may be fixed can be derived during the analysis phase or during program design. Alternatively, the programmer can rely on general software engineering knowledge to write specialization classes, as described in the next chapter.

All specialization classes written for a given usage scenario should be grouped together in a single *specialization module.* The finished program can be configured according to a specific usage scenario by specializing according to the corresponding specialization module. This use of specialization classes is akin to Boinot et al.'s adaptation classes, which is a framework inspired by specialization classes designed for creating adaptive programs [20]. Currently, the specialization class framework supports specialization modules by grouping together specialization classes in files. A set specialization modules are selected via the specialization class compiler command line, which causes the program to be specialized according to the specialization classes contained in these modules. While sufficient for simple scenarios with a few specialization modules, a more elaborate scheme is needed to handle more complex scenarios. However, we consider the development of such an extended system to be future work.

6.6 Summary

To reduce the complexity of applying partial evaluation, Volanschi et al. developed the specialization class framework. Specialization classes offer a uniform, declarative interface to controlling partial evaluation, and permit automatic re-integration of specialized code into the program under the control of automatically generated guards. We extend the specialization class language to encompass the features needed to specialize object-oriented programs using partial evaluation. In particular, our extended syntax allows one to specify abstract invariants that describe an arbitrary object structure. In addition, we define clear and unambiguous syntax and semantics for declaring object composition.

We consider specialization classes essential for the use of partial evaluation as a software engineering tool: they provide a simple and declarative syntax for the gradual introduction of specialization into a program as it is being developed. The aspect-like nature of specialization classes facilitates gradual introduction of specialization into a program without disturbing its implementation. Indeed, the creation of specialization classes during software development yields a finished program that can be automatically configured to specific usage scenarios.

Chapter 7

Specialization Patterns

7.1 Introduction

Design patterns offer numerous advantages in terms of software development; they convey knowledge about proven software designs and facilitate program adaptation. With design patterns, program adaptation is done both statically during program development and dynamically after deployment of the finished program [64]. Nonetheless, the use of design patterns in the development of a program can introduce inefficiency into the finished program. The potential for adaptability inherent in many design patterns is also present in the implementation, in the form of highly generic object-oriented designs that are systematically introduced into the program. When the potential for adaptation is greater than the concrete need for adaptation, the use of such generic designs becomes a recurring overhead throughout the program. This issue remains largely unaddressed in the design pattern community.

When the adaptability introduced by a design pattern is not needed in a specific usage context, specialization of the program to this specific usage before execution improves efficiency. As we have seen, such specialization can be done using object-oriented partial evaluation. Nevertheless, specialization is not always beneficial; for example specialization with respect to too many different forms of usage can cause code explosion. Therefore, the user must explicitly target the partial evaluator toward particular invariants and code regions. The specialization class framework discussed in the previous chapter provides a simple and declarative notation for expressing such specialization opportunities. However, the problem of identifying specialization opportunities remains. Profiling can help, but it may not reveal systematic structural overheads that block optimization throughout the program. A systematic approach taking into account the program design is needed.

We observe that the use of design patterns in a program gives rise to patterns of structural properties, which in turn give rise to patterns of overheads that form patterns of opportunities for specialization. We propose the use of *specialization patterns* as a complement to design patterns, to describe when, how, and exactly where a program structured using design patterns can benefit from specialization. This approach retains the program structuring advantages of design patterns, while relying on an automated transformation to map generic code into an efficient implementation.

The purpose of this chapter is to present specialization patterns. The declaration of *what* to specialize is addressed by specialization classes presented in Chapter 6 and the question of *how* to specialize is addressed with object-oriented partial evaluation described in Chapter 5. Specialization patterns address the key issue of selecting *where* to specialize.

Chapter overview: First, Section 7.2 explains the common pitfalls of applying partial evaluation. Then, Section 7.3 describes specialization of design patterns by means of specialization patterns. Section 7.4 presents a large example of how specialization patterns aid in specializing a complex program. Afterward, Section 7.5 assesses the utility of specialization patterns in the application of program specialization to uses of design patterns, and last Section 7.6 summarizes the contents of this chapter.

Note: Specialization patterns have been presented in earlier work by Schultz et al. [144]. This chapter is an extended version of this paper, which treats software engineering issues in more detail, and reflects the point of view that partial evaluation should be applied to a program during its development.

7.2 Partial Evaluation Pitfalls

Specialization of programs using partial evaluation should be guided by the user; partial evaluation must be directed towards critical program points that constitute interesting specialization opportunities, and must be carefully controlled to ensure an appropriate degree of specialization.

7.2.1 Directing partial evaluation

A partial evaluator cannot deduce specialization invariants from a large program as precisely as a human programmer. Partial evaluation must be targeted carefully since specialization of non-critical parts of a program may cause overheads. Thus, specialization is most effective when directed by the user to a limited part of the program where specialization is believed to be beneficial. A partial evaluator can be designed to operate on a program slice, which can be inserted back into the program after specialization. Such a program slice, and the invariants for which it is to be specialized, can be concisely described using specialization classes. In the context of design patterns, specialization classes allow the programmer to specialize for local invariants that only hold for the objects that play a role in the use of a design pattern.

7.2.2 Limitations of partial evaluation

The benefits of partial evaluation are limited by the degree to which the static information contained in the user-supplied invariants can be propagated throughout the program, and by the utility of the transformations triggered by these invariants. However, values computed in the program that depend on unknown information are considered dynamic and are not used in the specialization of the program. Consequently, partial evaluation is most effective when there is a clean separation in the program between the terms that depend only on known information and the unknown terms of the program. Specialization classes can be used to provide the partial evaluator with extra information about local invariants. But because of the need to react at run time when specialization invariants are invalidated, the use of specialization classes adds inefficiency. Excessive propagation of invariants can also be detrimental in different ways: specialization can cause code explosion either by too much loop unrolling, or by generating a very large number of specialized methods.

Because of these limitations, partial evaluation has to be carefully targeted by the user. We propose specialization patterns as a means to communicate the kinds of invariants that are worth declaring using specialization classes, and to document potential pitfalls in specializing a given design.

7.3 Specialization Patterns

Design patterns facilitate communication of design ideas by encapsulating a characterization of a common problem together with a solution to this problem into a single logical unit. Specialization patterns complement design patterns, by documenting a specialization process that results in an efficient implementation.

7.3.1 Specialization patterns: definition and use

A specialization pattern describes the overheads intrinsic in using a particular design pattern, and documents how to use partial evaluation to eliminate these overheads. In addition, a specialization pattern can refer to other specialization patterns, to describe how multiple design patterns can be specialized together. Specialization patterns not only guide specialization of a finished program, but can also help the programmer to structure the program so that specialization will be beneficial.

In the spirit of design patterns, a specialization pattern is described according to the template shown in Figure 7.1. The template includes sections that relate the specialization pattern to the design pattern, criteria to specialize a use of the design pattern, detailed instructions for performing specialization, and a specialization example. Examples of specialization patterns are given in the the remainder of this section.

Figure 7.2 shows how specialization patterns fit into the software development process. A generic program is written using design patterns where appropriate. To ensure that critical uses of design patterns can be subsequently specialized, specialization patterns are used in the development of the program. The specialization patterns describe how to use design patterns such that successful specialization is guaranteed, and how to write specialization classes that express this specialization. Once the program is written, the specialization classes corresponding to a specific usage of the program are selected based on

Name: The name of the associated design pattern.

Description: A short description of the design pattern.

Extent: The minimal program slice that is relevant when optimizing a use of the design pattern.

Overhead: Possible overheads associated with use of the design pattern.

Compiler: Analysis of when these overheads are eliminated by standard compilers.

Approach: Specialization strategies that eliminate the identified overheads.

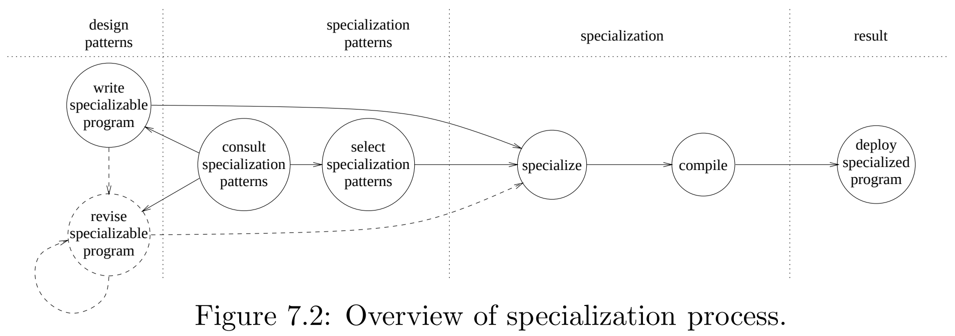
Condition: The conditions under which the specialization strategies can be effectively exploited.

Specialization class: Guidelines for how to write the needed specialization classes, and how to most effectively apply them.

Applicability: A rating of the overall applicability of specialization to a use of the design pattern, using the other information categories as criteria.

Example: An example of the use of specialization to eliminate the identified overhead; the example may include specialization classes or textual descriptions.

Figure 7.1: Specialization pattern template.


design patterns
specialization patterns
specialization
result
write specializable program
consult specialization patterns
select specialization patterns
specialize
compile
deploy specialized program
revise specializable program
Figure 7.2: Overview of specialization process.

what specialization patterns were used. The program and the specialization classes are then provided to the specializer, which generates a specialized program.

To illustrate the problems that can be addressed by specialization patterns, in the remainder of this section, we identify the specialization opportunities provided by *creational*, *structural*, and *behavioral* design patterns, and present examples of specialization patterns. The specialization example included with each specialization pattern has been structured with that specialization pattern in mind, which ensures that any intrinsic opportunities for specialization can be exploited using partial evaluation.

7.3.2 Creational patterns

A creational design pattern abstracts the construction of objects, known as the *products*, by delegating parts of the instantiation process to auxiliary classes. The use of a creational pattern separates the operations on an object from the underlying representation, which allows the representation to be changed transparently. Nevertheless, this abstraction barrier implies that the products must be accessed using virtual calls, which blocks optimization.

Memory allocation and object initialization dominate the cost of object creation, so specialization to only eliminate virtual calls associated with the creation process is unlikely to significantly optimize a program. Thus, the parts of the program where the products are used should also be specialized with respect to the concrete type of each product. Such specialization permits direct access to the products, which enables ordinary intraprocedural optimizations. However, such specialization is only effective when the specializer can determine how the products are manipulated after they have been created. This is outside the part of the program covered by the creational pattern, so a specialization pattern can only give limited information on when it is worthwhile to specialize.

Example: builder pattern

Figure 7.3a shows the `ListBuilder` interface for creating `AbstractList` lists using the builder pattern. An implementation must provide the methods `start`, which initializes the list, `add`, which extends the list, and `getProduct`, which finishes the production sequence by returning the list. Also defined is the class `Main` with a method `f`, which uses the `ListBuilder` interface to construct a list, and then accesses the `i`'th element of the list just produced. Figure 7.3b shows the concrete builder implementation `LinkedListBuilder`, which produces linked lists of type `LList`. The definitions of `AbstractList` and `LList` are shown in Figure 7.4.

Figure 7.5 defines the specialization pattern for the builder pattern. The specialization pattern suggests to specialize the program fragment with regards to a concrete builder implementation. The specialization class of Figure 7.3c specifies that the method `f` of the class `Main` should be specialized with respect to the `LinkedListBuilder` implementation, and in addition for a specific list index. In the specialized program (Figure 7.3d), virtual calls have been replaced by direct data-structure manipulations. Specialization replaces the virtual calls through the `ListBuilder` interface by direct calls, to which inlining is applied during specialization post-processing.

Specialization to a single concrete implementation permits the products to be accessed directly as long as they are not manipulated in a dynamic way. In the example, the product is used in a fixed way, and the virtual call to `lookup` has been replaced by a specialized version of its concrete definition in the `LList` class. If desired, the method `X.something` can also be specialized, adapting it to the concrete value stored as the second element of the `LList` object. Had the product been manipulated under the control of dynamic data, the benefits of specialization would have been negligible.

```
interface ListBuilder {
 void start();
 void add( Object o );
 AbstractList getProduct();
}
class Main {
 ListBuilder b;
 void f(int i) {
  b.start();
  b.add("x");
  b.add(new Vector());
  AbstractList p = b.getProduct();
  X.something(p.lookup(i));
 }
}
```

(a) Use of builder through interface

```
class LinkedListBuilder
      implements ListBuilder {
 LList head, tail;
 void start() { //add empty head
  head = new LList(null);
  tail = head;
 }
 void add( Object x ) {
  tail.next = new LList(x);
  tail = tail.next;
 }
 AbstractList getProduct() {
  return head.next; //discard head
 }
}
```

(b) Concrete builder for linked lists

```
specclass Main_LL specializes Main {
 b: LinkedListBuilder;
 @specialize:
  void f(int i) where i==1;
}
```

(c) Declaration of specialization to the LinkedListBuilder builder

```
aspect Main_LL {
 introduction Main {
  void f_LinkedListBuilder() {
   LinkedListBuilder b;
   b.head = new LList(null);
   b.tail = b.head;
   b.tail.next = new LList("x");
   b.tail = b.tail.next;
   b.tail.next =
     new LList(new Vector());
   b.tail = b.tail.next;
   AbstractList p = b.head.next;
   X.something(((LList)p).next.elm);
  }
 }
}
```

(d) Result of specialization

Figure 7.3: Specializing a use of the builder pattern.

Other creational patterns

In addition to the builder pattern, the abstract factory and prototype patterns also hide the types of the objects that they produce; thus, uses of these patterns are good targets for specialization. But as is the case for all creational patterns, whether the program will benefit from specialization depends on how the products are manipulated. The factory and singleton patterns are much simpler, and the types of the objects that they produce is usually evident. Uses of these patterns are thus easily handled by an optimizing compiler, but can of course be specialized as well.

7.3.3 Structural patterns

Structural design patterns organize relations between objects; this organization allows the programmer to combine individual objects that respect a common interface into compound objects that behave in new ways. By separating the objects using interfaces, structural patterns allow the object structure to be transparently extended, and new classes implementing the interface to be added. This flexibility implies, however, that the components must interact using virtual calls, which obscures the flow of control through the object structure,

```
interface AbstractList {
  Object lookup( int index );
  ... other methods defining AbstractList ...
}
class LList implements AbstractList {
  Object elm; LList next;
  LList( Object e ) { this.elm=e; }
  Object lookup( int i ) {
    return i==0?elm:next.lookup(i-1);
  }
  ... other methods for implementing AbstractList ...
}
```

Figure 7.4: Relevant parts of `AbstractList` and the implementation `LList`.

and impedes compiler optimizations.

A program that builds and traverses an object structure can be specialized to a specific layout of this structure. Specialization permits the objects to interact directly, and collects all of the basic operations on the structure in a single place. If the structure is not modified after its creation, the methods that traverse it can be directly specialized to its layout. When the structure is modifiable, specialization classes can be used to describe layouts that are of interest. As always, specialization classes introduce overheads, so the specialization class approach might not be beneficial if the structure changes too often, or if little time is spent actually traversing the structure. Because specialization of a structural pattern can generate code having size proportional to the size of the object structure, specialization should be applied with caution to avoid code explosion.

Example: bridge pattern

Figure 7.6a shows a use of the bridge pattern. The class `Complex` represents an interface object for complex numbers, with the specific implementation deferred to a `ComplexImpl` object. The `multiply` operation is delegated to the concrete implementation, whereas the `square` operation is defined in the interface object. The function `f` of the class `SquareFn` simply computes the square of a complex number. Two concrete implementations of `ComplexImpl` are given in Figure 7.6b: the `RectComp` implementation uses rectangular coordinates to represent complex numbers, whereas the `PolarComp` implementation uses polar coordinates.

Figure 7.7 defines the specialization pattern for the bridge pattern. The specialization pattern suggests to fix the type of the implementation object. The specialization class `SquareFn_RectComplex` (Figure 7.6c) specifies that `f` should be specialized to complex numbers that fulfill the invariants given in the `Rectangular` specialization class. This specialization class specifies that the implementation object has type `RectComp`. The result of specialization is an implementation of `f` where the mathematical operations are applied directly to the object fields. Specialization replaces virtual calls from `f` to the bridge

Name: Builder pattern

Description: The builder pattern allows a complex structure to be created by invoking a sequence of methods defined in a generic builder interface, thus separating the construction process from the underlying representation.

Extent: Specialization is applied to a collection of classes implementing the concrete representation of a structure, a class implementing the builder interface, and a client, which builds a structure using the generic operations provided by the builder interface. Specialization can also be applied to any subsequent use of the product structure.

Overhead: Separation of the type of the product from the client means that the product must be accessed using virtual calls.

Compiler: When there is either just a single kind of builder or a single kind of product, a compiler can usually generate direct calls for accessing the methods of the product. Nevertheless, a compiler typically does not make use of initialization information.

Approach: Specializing the client with respect to a particular implementation of the builder makes the objects comprising the structure directly accessible to the client. Accesses to the components of the structure can then be implemented using direct calls to the methods of these objects. Information about the current state of these objects can be used for further optimizations.

Condition: The type of the builder must be known to the specializer (possibly as a specialization class invariant). To guarantee specialization of the builder, the sequence of building actions must be fixed within the program. Furthermore, to guarantee direct use of the products and that information about their state is exploited by the specializer, they must be used in a fixed way.

Specialization class: The specialization class should fix the type of the builder, and specify specialization of a method that both uses the builder and the resulting product.

Applicability: High when the specialization class can be placed properly and the products are used often. Low to none otherwise.

Example: See Figure 7.3 and explanation in text.

Figure 7.5: Builder specialization pattern.

interface object by direct calls, and similarly replaces virtual calls from the interface object to the implementation object by direct calls. These direct calls are inlined, eliminating temporary variables when appropriate, to produce the specialized implementation of `f` shown in Figure 7.6d.

Other structural patterns

The adapter, composite, decorator, facade, and proxy structural patterns also build structures from objects hidden behind generic interfaces, so uses of these patterns are good targets for specialization. Specialization is guaranteed to simplify the program when the structure does not change or when it can be encap-

```
interface ComplexImpl {
 void mult( Complex c );
 double r();
 double i();
}
class Complex {
 ComplexImpl imp;
 void multiply( Complex c ) {
  imp.mult( c );
 }
 void square() {
  imp.mult( this );
 }
 double r() { return imp.r(); }
 double i() { return imp.i(); }
}
class SquareFn {
 void f( Complex x ) {
  x.square();
 }
}
```

(a) Generic number interface and use

```
class RectComp
      implements ComplexImpl {
 double r, i;
 void mult( Complex c ) {
  double cr = c.r();
  double ci = c.i();
  double nr = r*cr - i*ci;
  double ni = r*ci + i*cr;
  r = nr; i = ni;
 }
 double r() { return r; }
 double i() { return i; }
}
class PolarComp
      implements ComplexImpl {
  ...polar coordinates...
}
```

(b) Specific number implementations

```
specclass SquareFn_RectComplex
         specializes SquareFn {
 @specialize:
  void f( Complex x ) where
         x: Rectangular;
}
specclass Rectangular
          specializes Complex {
 imp: RectComp;
}
```

(c) Declaration of specialization to the `NormalNum` implementation

```
aspect RectSquare {
 introduction SquareFn {
  void f_RectComplex( Complex x ) {
   RectComp tmp = (RectComp)x.imp;
   double cr = tmp.r;
   double ci = tmp.i;
   double nr = tmp.r*cr - tmp.i*ci;
   double ni = tmp.r*ci + tmp.i*cr;
   tmp.r = nr; tmp.i = ni;
  }
 }
}
```

(d) Result of specialization

Figure 7.6: Specializing a use of the bridge pattern.

sulated using specialization classes. The flyweight pattern optimizes memory usage by sharing objects, and cannot be specialized in any obvious way.

7.3.4 Behavioral patterns

Behavioral patterns abstract over the control flow, providing generic ways of parameterizing behavior. They separate different aspects of an overall behavior, which makes it possible to construct new behaviors by composing individual objects or classes. Every time the collaborating objects are used for a specific function, they must interact with each other using virtual calls.

A program using a behavioral pattern can be specialized to a specific behavior, by specifying the values and objects that control the behavioral pattern. Specialization transforms the complete description of the behavior into a single unit. Nevertheless, the behavioral design patterns are so diverse that it is only for specific patterns that we can guarantee benefits from specialization. Depending on the specific pattern in question, specialization can be done by specializing the pattern use to the object structure that it processes, and possibly to any values that control how it processes the object structure. In any

Name: Bridge pattern

Description: The bridge pattern separates an object into a generic interface object and an implementation object. The client accesses the object via the generic interface object, which provides access to a specific implementation object using delegation. The bridge pattern decouples an abstraction from its implementation, so that the two can vary independently.

Extent: Specialization is applied to an interface object which has operations that are defined in terms of generic operations on an implementation object.

Overhead: To allow the implementation to vary independently from the generic interface object, method invocations from the generic interface object to the implementation object are implemented using virtual calls. Furthermore, the client typically requires access to the methods of the implementation object, which implies two levels of method invocation (first a call to the interface object, then the subsequent call to the implementation object).

Compiler: Since there normally are multiple implementation classes, a compiler can rarely eliminate the virtual call from the interface object to the implementation object.

Approach: When the components connected by the bridge are known, the bridge can often be eliminated. This allows direct access from the interface object to the implementation object.

Condition: If the coupling between the interface object and the implementation object never changes or only changes in a fixed way, then it can be specialized to remove one level of indirection. If the interface object is used by the program in a fixed way, then the interface indirection can be specialized away as well.

Specialization class: The specialization class should at least fix the type of the implementation object, and should ideally specify specialization of a method that uses the interface object several times. Alternatively, when a single method in the interface object is implemented using several calls to methods in the implementation object, the specialization class can specify specialization of the interface object.

Applicability: Medium when the specialization class can be targeted properly (see previous item) or when there are many calls from the interface to the implementation object. Low otherwise.

Example: See Figure 7.6 and explanation in text.

Figure 7.7: Bridge specialization pattern.

case, if the objects that make up the use of the pattern cannot be determined by the specializer, the behavioral pattern cannot in general be specialized.

Example: strategy pattern

Figure 7.8a shows a use of the strategy pattern. The `Image` class represents an image using pixels defined by the `RGB` class. The `process` method of an `Image` object applies the pixelwise processing strategy stored in the field `op` to

```
interface RGBOP {
 void handle( RGB pixel );
}
class RGB { double r, g, b; }
class Image {
 RGB [][]img; int w, h;
 RGBOP op;
 void process() {
  for(int i=0; i<w; i++)
   for(int j=0; j<h; j++)
    op.handle(img[i][j]);
 }
 ...
}
```

(a) RGB Image which uses operator

```
class Scale implements RGBOP {
 double s;
 void handle( RGB p ) {
  p.r*=s;p.g*=s;p.b*=s;
 }
}
class RedOnly implements RGBOP {
 ...keep only red component...
}
```

(b) RGB pixel operations

```
specclass ScaleByTwoProcess
          specializes Image {
 op: ScaleByTwo;
 @specialze: void process();
}
specclass ScaleByTwo
          specializes Scale {
 s == 2.0;
}
```

(c) Declaration of specialization to the `Scale` operation

```
aspect ScaleByTwo {
 introduction Image {
  void process_ScaleByTwo() {
   for(int i=0; i<w; i++ )
    for(int j=0; j<h; j++ ) {
     RGB p = img[i][j];
     p.r*=2.0;p.g*=2.0;p.b*=2.0;
    }
  }
 }
}
```

(d) Result of specialization

Figure 7.8: Specializing a use of the strategy pattern.

each pixel of the image. Figure 7.8b defines two such single-pixel operations: `Scale`, which scales a pixel (thereby changing its brightness), and `RedOnly`, which discards all but the red component.

Figure 7.9 defines the specialization pattern for the strategy pattern. The specialization pattern suggests to specialize for a specific algorithm. The specialization class `ScaleByTwoProcess` (Figure 7.8c) declares that the operation is a `Scale` operation, and that the scaling value is 2.0. We thus specialize the `Image` class to a strategy that is specified not only in terms of its type, but also in terms of its internal state. Specialization merges the effect of the strategy object into the original `process` method (Figure 7.8d), by eliminating the virtual call to the strategy method, inlining the call, and propagating the known scaling value.

Other behavioral patterns

As is the case for the strategy pattern, precise specialization patterns can be given to the chain of responsibility, interpreter, mediator, observer, state, and visitor patterns. For the interpreter and visitor patterns, specialization is beneficial when the use of the pattern can be specialized with respect to the structure processed by the pattern, in which case the use of the pattern can be completely eliminated. The command and iterator design patterns represent opportunities for specialization, but it is difficult to precisely specify when this is the case, except for the most basic case where the behavior is completely fixed. The

Name: Strategy pattern

Description: The strategy pattern allows clients to transparently replace one algorithm by another. This design pattern allows clients to choose among whole families of algorithms rather than just a single algorithm.

Extent: Specialization is applied to a family of algorithms all implementing the same abstract interface, and a client that uses such algorithms through this abstract interface.

Overhead: All operations provided by the algorithm must be accessed through the abstract interface. The less computation is done by the algorithm, the more this overhead is noticeable.

Compiler: Unless the strategy is chosen explicitly before it is used, a compiler is unlikely to bypass the abstract interface.

Approach: By specializing the client to the concrete algorithm, the abstract interface can be bypassed, and the algorithm can be inlined into the client. This opens opportunities for further specialization and optimization of the algorithm to the context in which it is being used.

Condition: If the coupling between the client and the concrete strategy being used never changes, then the client can be specialized to this strategy. If the coupling does not change during the invocation of a method in the client, the specialization classes can introduce a local invariant (as in the example), allowing this method to be specialized to the strategy.

Specialization class: The specialization class should fix the type of the strategy, and specify specialization of a method that applies the strategy.

Applicability: High when the strategy is used inside a loop, medium when used a few times, low when used only once.

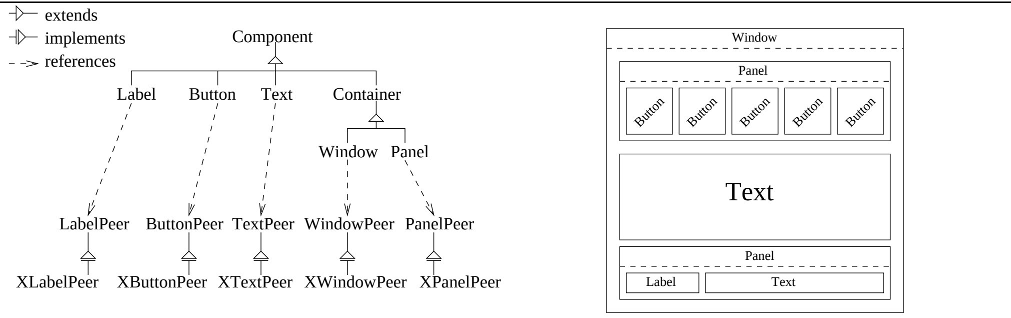
Example: See Figure 7.8 and explanation in text.

Figure 7.9: Strategy specialization pattern.

template method pattern obtains genericity through inheritance, and can easily be handled by an optimizing compiler. The memento pattern externalizes the state of an object, and cannot be specialized in any general way.

7.4 A Complete Example

To illustrate the combined effects of specialization of several design patterns, we provide a complete example: a graphical application. We first describe the example, with a focus on the toolkit used to write the application. Then, we characterize the overheads present in the application and explain how they can be eliminated.



(a) Class diagram (b) Structure of the application

Figure 7.10: Overview of graphical application.

7.4.1 Description

Our example is a graphical text editor application written using an abstract windowing toolkit, styled after the Java JDK 1.1 AWT (abstract windowing toolkit). Graphical windowing toolkits often contain many opportunities for specialization, and the JDK 1.1 AWT is no exception.

We focus on the following uses of design patterns in the toolkit:

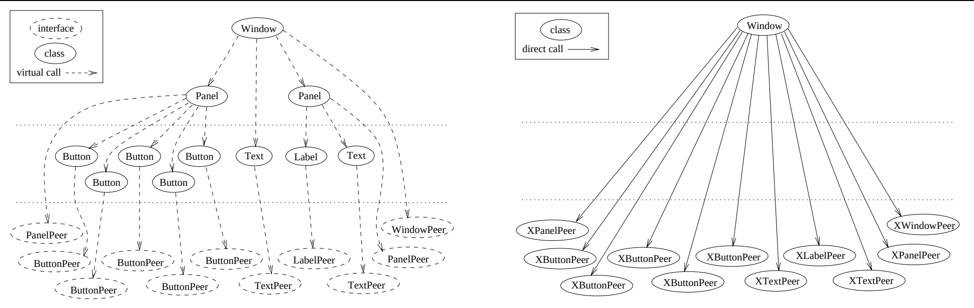
- The structural pattern *composite* is central to most graphical toolkits. It allows graphical widgets and widget containers to be freely combined. To use the composite pattern, each graphical widget and container extends an abstract class, `Component` (the name used in JDK 1.1).
- To allow our toolkit to function with any concrete windowing environment, we separate each component into its general representation and its system-specific *peer*, by using the bridge pattern. The peer objects are created using the *abstract factory* creational pattern, which defines a general interface for creating peers. To use the abstract factory, we let a central class (named `Toolkit` in JDK 1.1) function as an interface for instantiating peers.
- To simplify event handling, we use the *observer* behavioral pattern (for which there are standard interfaces in JDK 1.1). Clients subscribe to events generated by specific widget objects, such as buttons, and are notified when these events occur using standard Java method calls.

The class diagram of the basic graphical objects of the toolkit is shown in Figure 7.10a.

When launched, the text editor initializes itself, arranging its graphical appearance (illustrated in Figure 7.10b), and then waits for input events to be generated by the user, each event triggering a specific action.

7.4.2 Overheads

Each use of a design pattern in the graphical toolkit gives rise to a specific overhead. The use of the composite pattern makes it possible to change the



(a) Before specialization (b) After specialization

Figure 7.11: Call graph of `repaint()`, before and after specialization.

graphical appearance of the program at run time. A virtual method such as `repaint`, which is implemented by all the `Component` objects, must use virtual calls to traverse the composite structure. The use of the abstract factory (together with the bridge) hides the types of the peer objects, allowing new peers to be introduced at any point; all calls from a `Component` object to its native implementation are thus virtual. Finally, every time an event is generated, a corresponding event object is created and passed to the observers currently subscribed to this event. Each observer inspects the event and acts accordingly.

7.4.3 Specialization

First, the application is specialized to the way the composite objects are composed. The `repaint` method of the root `Window` object can thus refresh the entire application at once, without traversing the widget structure. Next, the application is specialized to the concrete peer objects being used. This transformation allows composite objects to directly manipulate their peer objects. Finally, the application is specialized to the concrete observer/observee relations of the application. An event now results in the actions that it implies being directly performed throughout the application.

Figure 7.11a shows the call graph of invoking the generic `repaint` method on the top-level window object. The object structure is traversed, using virtual calls to propagate the `repaint` operation to all the peer objects. By contrast, Figure 7.11b shows the call graph of invoking the `repaint` method after specialization to the widget structure and a specific set of peers. Calls are made directly to the peer objects, without traversing the object structure.

A more complex application might manipulate its graphical appearance while running. However, such an application would often be divided into several independent units, that are configured individually. Specialization can be applied separately to each such unit.

7.5 Assessment

We have argued that partial evaluation is difficult to control. We present specialization patterns as a solution to this problem, and argue for their utility based on specialization examples. However, to be useful in the software development process, specialization patterns must offer advantages both in terms of software engineering and performance.

7.5.1 Software engineering and specialization patterns

Since their conception, design patterns have gained widespread popularity, and many programs are structured according to design pattern principles. The more common the use of design patterns, the more valuable the specialization pattern approach: specialization patterns provide immediate insights as to how programs written according to design pattern principles can be specialized.

Due to space constraints and the lack of widespread practical experience in working with design patterns, the specialization patterns presented in this chapter are limited in their number, their level of detail, and the amount of information that they convey. Hence, the information that they provide is primarily useful to the novice partial evaluation user. Nonetheless, specialization patterns communicate useful intuition on how object-oriented programs can be specialized using partial evaluation. With the development of more detailed specialization patterns, that include previous experience with specialization, detailed benchmarks, extensive examples, and more descriptions of where each specialization strategy works well, specialization patterns should prove genuinely useful to a would-be object-oriented partial evaluation community.

Specialization patterns could even find a wider use, since design patterns exist for functional and imperative languages. Specialization patterns could be conceived for these languages to describe specialization of language-specific designs. However, design patterns have thus far mostly been restricted to object-oriented languages; it seems likely that the continued evolution of specialization patterns may be restricted to object-oriented languages.

7.5.2 Performance issues

The specialization strategies described by specialization patterns are targeted toward eliminating abstraction barriers erected by the use of design patterns. The elimination of these barriers optimizes the program, but only to a certain extent. The running time of the program may be dominated by other computations, or there may be other opportunities for specialization that when exploited would give a larger speedup. Nonetheless, it is often the specialization of a use of a design pattern and the ensuing program structure simplification that enables further specialization or optimization of the program.

To illustrate the performance benefits of eliminating uses of design patterns by specialization, Chapter 11 presents complete benchmarks based on the examples of Section 7.3. In the benchmarks reported, the speedup due to specialization varies with the complexity of the adaptation taking place in the benchmark, from almost nothing to a four-fold speedup. In practice, however, the improvement due to specialization can vary widely, depending on the number of specialization opportunities introduced by eliminating the abstraction barriers created by the use of design patterns.

7.6 Summary

Design patterns focus on how programs should be structured to offer features such as modularity and extensibility. However, this structuring is directly mapped into an implementation; features are directly implemented in terms of mechanisms that cause overheads at run time. Still, these overheads are predictable because they are inherent to each design pattern. Specialization patterns is an approach aimed at optimizing patterns of overheads identified in design patterns. This optimization process, based on partial evaluation, removes abstraction layers by exploiting information about object delegation. This approach is applicable to several kinds of design patterns (creational, structural, and behavioral): for each kind of design pattern, we have characterized specialization opportunities, and used examples to concretely show the effectiveness of program specialization in removing the overheads inherent to design patterns.

In effect, partial evaluation can be used systematically to map programs developed using design patterns into efficient implementations. This mapping is guided by information provided by design patterns. As a result, we have extended the scope of design patterns: not only do they guide program development, but they also enable systematic optimization of the resulting programs.

Part III

Object-Oriented Partial Evaluation

Chapter 8

Formalization of Object-Oriented Partial Evaluation

8.1 Introduction

The first partial evaluators were constructed in an ad-hoc fashion using a mix of standard program analysis and transformation techniques [72, 73]; it was with the advent of self-applicable, off-line partial evaluators that the effect of partial evaluation on a program was precisely specified in terms of a formalized model [87]. Although the complexity of a partial evaluator implementation often increases with the complexity of the target programs, a formalization remains a concise and useful specification of its behavior.

This chapter defines partial evaluation for object-oriented languages through a formal description of a partial evaluator for a small object-oriented language. We describe how to perform a binding-time analysis, and how to subsequently specialize a binding-time annotated program. The partial evaluator defined in this chapter is intended to be minimal; in the next chapter we discuss what features are needed for a partial evaluator to target realistic applications. We work with class-based languages throughout this chapter, except in Section 8.8 where we describe partial evaluation for object-based languages.

Chapter overview: First, Section 8.2 provides a formalization of the small Java-like language presented in Chapter 2. Then, Section 8.3 extends this language to a two-level language, and Section 8.4 defines typing rules for the two-level separation of a program. Section 8.5 defines specialization in terms of evaluation of two-level programs, and Section 8.6 defines a binding-time analysis for automatically deriving a two-level program. Afterward, Section 8.7 gives examples of how our partial evaluator specializes object-oriented programs. Last, Section 8.8 describes partial evaluation for object-based languages, and Section 8.9 presents a summary.

8.2 Language: Extended Featherweight Java

To provide a thorough yet concise description of partial evaluation for class-based object-oriented languages, we provide a formal definition of the EFJ language that was informally introduced in Chapter 2. In this section, we investigate language features that introduce key aspects of partial evaluation, and then provide syntax, semantics, and typing rules for the resulting language.

8.2.1 Language concerns

The choice of what programming language features to include in EFJ is influenced by what features are found in common object-oriented languages, and what features are interesting in terms of partial evaluation. Encapsulation of data and methods are essential object-oriented features, and are included in the language. EFJ is a class-based language, so standard inheritance between classes is also included. Almost all object-oriented languages are imperative; the idea of an autonomous self-updating object is most easily captured in an imperative language. Nevertheless, we omit imperative features from EFJ, since it is well-known how to handle imperative features in a partial evaluator [6, 9, 41]. Also, most object-oriented languages have an eager semantics, which we also choose for EFJ. The inclusion of standard language features such as conditionals, operators, and boolean and integer constants makes writing examples easier. So while they are not specific to object-oriented languages and it is well-known how to partially evaluate them, we include them in EFJ.

Compared to the original Featherweight Java (FJ) language of Igarishi et al.[81], EFJ extends FJ with base type values (integers, booleans) and a conditional form. Where FJ has a non-deterministic small-step semantics, EFJ has a deterministic big-step semantics. Intuitively, EFJ is a subset of Java, and any EFJ program behaves like the syntactically equivalent Java program. The only exception is that EFJ boolean operators (`&&` and `||`) are eager, whereas the corresponding Java operators short-circuit evaluation.

8.2.2 EFJ syntax

The syntax of EFJ is given in Figure 8.1. A program is a collection of classes and a main expression. Each class in the program extends some superclass, declares a number of fields, a constructor, and a number of methods. Fields are either of base type or object (reference) type. The constructor always calls the constructor of the superclass first and then initializes each field declared in the class afterward; the constructor is the only place where fields can be assigned values. As is the case in Featherweight Java, the appearance of a constructor is fixed given the fields of a class and its superclass. The semantics of object initialization is not defined in terms of the constructor but is defined directly in terms of the fields of the class. However, writing out the constructor allows us to retain a Java-compatible syntax. A method declares its formal parameters and has a body consisting of a single expression. An expression can be a constant, a variable, a field lookup, a virtual method invocation, an object instantiation,

P ∈ Program	::=	`({CL`$_1$`,...,CL`$_n$`},e)`
CL ∈ Class	::=	`class C extends C {T`$_1$` f`$_1$`;...;T`$_n$` f`$_n$`; K M`$_1$`...M`$_k$`}`
T ∈ Type	::=	`int` | `boolean` | `C`
K ∈ Constructor	::=	`C(T`$_1$` f`$_1$`,...,T`$_n$` f`$_n$`)`
		`{super(f`$_1$`,...,f`$_i$`); this.f`$_{i+1}$` = f`$_{i+1}$`;...;this.f`$_n$` = f`$_n$`;}`
M ∈ Method	::=	`T m(T`$_1$` x`$_1$`,...,T`$_n$` x`$_n$`) {return e;}`
e ∈ Expression	::=	`c` | `x` | `e.f` |`e.m(e`$_1$`,...,e`$_n$`)`
	|	`new C(e`$_1$`,...,e`$_n$`)` | `(C)e` | `e OP e` | `(e?e:e)`
OP ∈ Operator	::=	`+` | `-` | `*` | `/` | `<` | `>` | `==` | `&&` | `||`
c ∈ Constant	::=	`true` | `false` | `0` | `1` | `-1` | `...`
Values that result from computation:		
v ∈ Value	::=	`c` | object$_{\mathsf{C}}$`(v`$_1$`,...,v`$_n$`)`

Figure 8.1: EFJ syntax (program and values)

a class cast, a binary operator, or a conditional. As shown in Figure 8.1, values can either be constants or objects. An object value is a tuple of computed values that is labeled with the name of the class of the object.

The special class `Object` cannot be declared but is part of every program. This class extends no other class, and has no methods and no fields; with the exception of this class, all classes referenced in the program must also be defined in the program. Furthermore, there should be no cycles in the inheritance relation between classes.

In the other chapters of this document a program is represented by a number of classes and no main expression (there is assumed to be a main method). To simplify the formalization and examples presented in this chapter, we here use a version of EFJ with a main expression. We permit free variables in the main expression as a means of parameterizing a program.

Syntactic conventions: Throughout this chapter, the metavariables `T` and `R` range over types; `A`, `B`, `C`, `D`, and `E` range over class names; `f` and `g` range over field names; `m` ranges over method names; `x` ranges over parameter names; `d` and `e` range over expressions; `CL` ranges over class declarations; `K` ranges over constructor declarations; `M` ranges over method declarations; `c` ranges over constants; `OP` ranges over binary operators; and `v` ranges over computed values. Unlike the original presentation of Featherweight Java, we do not use the notation $\overline{\mathtt{x}}$ as a shorthand for $\mathtt{x}_1, \ldots, \mathtt{x}_n$. Sequences are always written out, to avoid confusion with underlining, which we use to indicate binding times.

Auxiliary definitions: The definitions in Figure 8.2 are used to extract information from the program; they are used throughout the chapter. The function *CL* maps a class name to its definition, the function *fields* maps a class name to a list of its fields, the function *mtype* maps a method name and class name to the method type, the function *mbody* maps a method name and a class name to the formal parameters and body of this method, and the function *override* checks whether a method override is legal (see next section). As is the case for the original FJ presentation, we have chosen the notion of a “current program”

Class table lookup:

$$\frac{(\{\mathtt{CL}_1,\ldots,\mathtt{CL}_n\},\mathtt{e})\text{ is current program}\qquad \mathtt{class\ C\ extends\ D\ \{\ \ldots\}} \in \{\mathtt{CL}_1,\ldots,\mathtt{CL}_n\}}{\mathit{CT}(\mathtt{C}) = \mathtt{class\ C\ extends\ D\ \{\ \ldots\}}}$$

Field lookup:

$$\mathit{fields}(\mathtt{Object}) = \epsilon$$

$$\frac{\mathit{CT}(\mathtt{C}) = \mathtt{class\ C\ extends\ D}\ \{\ \mathtt{C}_1\ \mathtt{f}_1;\ldots;\mathtt{C}_n\ \mathtt{f}_n\ \ldots\}\qquad \mathit{fields}(\mathtt{D}) = \mathtt{D}_1\ \mathtt{g}_1,\ldots,\mathtt{D}_n\ \mathtt{g}_n}{\mathit{fields}(\mathtt{C}) = \mathtt{D}_1\ \mathtt{g}_1,\ldots,\mathtt{D}_n\ \mathtt{g}_n,\mathtt{C}_1\ \mathtt{f}_1,\ldots,\mathtt{C}_n\ \mathtt{f}_n}$$

Method type lookup

$$\frac{\mathit{CT}(\mathtt{C}) = \mathtt{class\ C\ extends\ D}\ \{\ \ldots\ \mathtt{K}\ \mathtt{M}_1\ldots\mathtt{M}_n\ \}\qquad \mathtt{T\ m}(\mathtt{T}_1\ \mathtt{x}_1,\ldots,\mathtt{T}_k\ \mathtt{x}_k)\ \{\ \mathtt{return\ e;}\ \}}{\mathit{mtype}(\mathtt{m},\mathtt{C}) = \mathtt{T}_1,\ldots,\mathtt{T}_k \rightarrow T}$$

$$\frac{\mathit{CT}(\mathtt{C}) = \mathtt{class\ C\ extends\ D}\ \{\ \ldots\mathtt{M}_1\ldots\mathtt{M}_n\ \}\qquad \mathtt{m}\text{ is not defined in }\mathtt{M}_1,\ldots,\mathtt{M}_n}{\mathit{mtype}(\mathtt{m},\mathtt{C}) = \mathit{mtype}(\mathtt{m},\mathtt{D})}$$

Method body lookup:

$$\frac{\mathit{CT}(\mathtt{C}) = \mathtt{class\ C\ extends\ D}\ \{\ \ldots\mathtt{M}_1\ldots\mathtt{M}_n\ \}\qquad \mathtt{T\ m}(\mathtt{T}_1\ \mathtt{x}_1,\ldots,\mathtt{T}_k\ \mathtt{x}_k)\ \{\ \mathtt{return\ e;}\ \}}{\mathit{mbody}(\mathtt{m},\mathtt{C}) = ((\mathtt{x}_1,\ldots,\mathtt{x}_k),\mathtt{e})}$$

$$\frac{\mathit{CT}(\mathtt{C}) = \mathtt{class\ C\ extends\ D}\ \{\ \ldots\mathtt{M}_1\ldots\mathtt{M}_n\ \}\qquad \mathtt{m}\text{ is not defined in }\mathtt{M}_1,\ldots,\mathtt{M}_n}{\mathit{mbody}(\mathtt{m},\mathtt{C}) = \mathit{mbody}(\mathtt{m},\mathtt{D})}$$

Valid method overriding:

$$\frac{\mathit{mtype}(\mathtt{m},\mathtt{C}) = \mathtt{T}_1,\ldots,\mathtt{T}_n \rightarrow \mathtt{T}_0}{\mathit{override}(\mathtt{m},\mathtt{C},\mathtt{T}_1,\ldots,\mathtt{T}_n \rightarrow \mathtt{T}_0)}$$

Figure 8.2: EFJ auxiliary definitions

$$\mathtt{T}\ <:\ \mathtt{T} \qquad \text{(S-ID)}$$

$$\frac{\mathtt{C}\ <:\ \mathtt{D}\qquad \mathtt{D}\ <:\ \mathtt{E}}{\mathtt{C}\ <:\ \mathtt{E}} \qquad \text{(S-TRANS)}$$

$$\frac{\mathit{CT}(\mathtt{C}) = \mathtt{class\ C\ extends\ D\ \{\ldots\}}}{\mathtt{C}\ <:\ \mathtt{D}} \qquad \text{(S-CLASS)}$$

Figure 8.3: EFJ subtyping

to avoid threading the program definition through all rules.

8.2.3 EFJ typing

Like Java, EFJ is a statically-typed object-oriented language. The typing rules ensure that reduction never goes wrong: a well-typed program either reduces to a value, stops at an illegal type cast, or diverges. A type cast is illegal when an object of a more general type is cast to a more specific type, for example after being extracted from a data structure.

The subtyping relation used in EFJ (written <:) is defined in Figure 8.3. A type is a sub-type of itself (rule S-ID), subtyping is transitive for class types (rule S-TRANS), and the type defined by a class is a subtype of the type defined by its superclass (rule S-CLASS).

The typing rules for EFJ expressions, method declarations, and class declarations are defined in Figure 8.4. An environment Γ is a finite mapping from

Expressions:

$$\dfrac{\texttt{c} \in \{\texttt{true}, \texttt{false}\}}{\Gamma \vdash \texttt{c} : \texttt{boolean}} \quad (\text{T-Bool}) \qquad \dfrac{\texttt{c} : \{\texttt{0}, -\texttt{1}, -\texttt{1}, \ldots\}}{\Gamma \vdash \texttt{c} : \texttt{int}} \quad (\text{T-Int})$$

$$\Gamma \vdash \texttt{x} : \Gamma(\texttt{x}) \quad (\text{T-Var})$$

$$\dfrac{\Gamma \vdash \texttt{e}_0 : \texttt{int} \quad \Gamma \vdash \texttt{e}_1 : \texttt{int} \quad \texttt{OP} : \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}\}}{\Gamma \vdash \texttt{e}_0 \ \texttt{OP} \ \texttt{e}_1 : \texttt{int}} \quad (\text{T-OP-I-I})$$

$$\dfrac{\Gamma \vdash \texttt{e}_0 : \texttt{int} \quad \Gamma \vdash \texttt{e}_1 : \texttt{int} \quad \texttt{OP} : \{\texttt{<}, \texttt{>}, \texttt{==}\}}{\Gamma \vdash \texttt{e}_0 \ \texttt{OP} \ \texttt{e}_1 : \texttt{boolean}} \quad (\text{T-OP-I-B})$$

$$\dfrac{\Gamma \vdash \texttt{e}_0 : \texttt{boolean} \quad \Gamma \vdash \texttt{e}_1 : \texttt{boolean} \quad \texttt{OP} : \{\texttt{\&\&}, \texttt{||}, \texttt{==}\}}{\Gamma \vdash \texttt{e}_0 \ \texttt{OP} \ \texttt{e}_1 : \texttt{boolean}} \quad (\text{T-OP-B-B})$$

$$\dfrac{\Gamma \vdash \texttt{e}_0 : \texttt{boolean} \qquad \Gamma \vdash \texttt{e}_1, \texttt{e}_2 : \texttt{T}}{(\texttt{e}_0 ? \texttt{e}_1 \texttt{:}\ \texttt{e}_2) : \texttt{T}} \quad (\text{T-Cond})$$

$$\dfrac{\Gamma \vdash \texttt{e}_0 : \texttt{C}_0 \qquad \mathit{fields}(\texttt{C}_0) = \texttt{T}_1 \ \texttt{f}_1, \ldots, \texttt{T}_n \ \texttt{f}_n}{\Gamma \vdash \texttt{e}_0.\texttt{f}_i : \texttt{T}_i} \quad (\text{T-Field})$$

$$\dfrac{\Gamma \vdash \texttt{e}_0 : \texttt{C}_0 \quad \mathit{mtype}(\texttt{m}, \texttt{C}_0) = \texttt{T}_1, \ldots, \texttt{T}_n \to \texttt{T} \quad \Gamma \vdash \texttt{e}_i : \texttt{R}_i \quad \texttt{R}_i \ \texttt{<:} \ \texttt{T}_i}{\Gamma \vdash \texttt{e}_0.\texttt{m}(\texttt{e}_1, \ldots, \texttt{e}_n) : \texttt{T}} \quad (\text{T-Invk})$$

$$\dfrac{\mathit{fields}(\texttt{C}) = \texttt{T}_1 \ \texttt{f}_1, \ldots, \texttt{T}_n \ \texttt{f}_n \qquad \Gamma \vdash \texttt{e}_i : \texttt{R}_i \qquad \texttt{R}_i \ \texttt{<:} \ \texttt{T}_i}{\Gamma \vdash \texttt{new C}(\texttt{e}_1, \ldots, \texttt{e}_n) : \texttt{C}} \quad (\text{T-New})$$

$$\dfrac{\Gamma \vdash \texttt{e}_0 : \texttt{D} \qquad \texttt{D} \ \texttt{<:} \ \texttt{C} \ \vee \ \texttt{C} \ \texttt{<:} \ \texttt{D}}{\Gamma \vdash (\texttt{C})\texttt{e}_0 : \texttt{C}} \quad (\text{T-Cast})$$

Methods:

$$\dfrac{\begin{array}{c} \texttt{x}_i : \texttt{T}_i, \texttt{this} : \texttt{C} \vdash \texttt{e}_0 : \texttt{E}_0 \qquad \texttt{E}_0 \ \texttt{<:} \ \texttt{T}_0 \\ \mathit{CT}(\texttt{C}) = \texttt{class C extends D } \{\ldots\} \qquad \mathit{override}(\texttt{m}, D, \texttt{T}_1, \ldots, \texttt{T}_n \to \texttt{T}_0) \end{array}}{\texttt{T}_0 \ \texttt{m}(\texttt{T}_1 \ \texttt{x}_1, \ldots, \texttt{T}_n \ \texttt{x}_n) \ \{ \ \texttt{return e}_0\texttt{;} \ \} \ \texttt{OK IN C}}$$

Classes:

$$\dfrac{\begin{array}{c} \texttt{K} = \texttt{C}(\texttt{T}_1 \ \texttt{g}_1, \ldots, \texttt{T}_n \ \texttt{g}_n, \texttt{R}_1 \ \texttt{f}_1, \ldots, \texttt{R}_k \ \texttt{f}_k) \ \{ \ \text{super}(\texttt{g}_1, \ldots, \texttt{g}_n); \ \texttt{this.f}_1\texttt{=} \ \texttt{f}_1; \ldots; \texttt{this.f}_k\texttt{=} \ \texttt{f}_k \ \} \\ \mathit{fields}(\texttt{D}) = \texttt{T}_1 \ \texttt{g}_1, \ldots, \texttt{T}_n \ \texttt{g}_n \qquad \texttt{M}_i \ \texttt{OK IN C} \end{array}}{\texttt{class C extends D } \{ \ \texttt{C}_1 \ \texttt{f}_1; \ldots; \texttt{C}_n \ \texttt{f}_n \ \texttt{K} \ \texttt{M}_1 \ \ldots \ \texttt{M}_p \ \} \ \texttt{OK}}$$

Program:

$$\dfrac{\forall \texttt{CL} \in \{\texttt{CL}_1, \ldots, \texttt{CL}_n\} : \texttt{CL OK} \qquad \Gamma_0 \vdash \texttt{e} : \texttt{T}}{(\{\texttt{CL}_1, \ldots, \texttt{CL}_n\}, \texttt{e}) \ \texttt{OK IN} \ \Gamma_0}$$

Figure 8.4: EFJ typing

$$\frac{\texttt{e}_i \longrightarrow \texttt{v}_i}{\texttt{new C(}\texttt{e}_1,\ldots,\texttt{e}_n\texttt{)} \longrightarrow \mathsf{object}_{\texttt{C}}(\texttt{v}_1,\ldots,\texttt{v}_n)} \qquad \text{(R-New)}$$

$$\frac{\texttt{e} \longrightarrow \mathsf{object}_{\texttt{C}}(\texttt{v}_1,\ldots,\texttt{v}_n) \qquad \mathit{fields}(\texttt{C}) = \texttt{T}_1\ \texttt{f}_1,\ldots,\texttt{T}_n\ \texttt{f}_n}{\texttt{e.f}_i \longrightarrow \texttt{v}_i} \qquad \text{(R-Field)}$$

$$\frac{\begin{array}{c}\texttt{e} \longrightarrow \mathsf{object}_{\texttt{C}}(\texttt{v}_1,\ldots,\texttt{v}_n) \qquad \texttt{d}_i \longrightarrow \texttt{d}'_i \\ \mathit{mbody}(\texttt{C},\texttt{m}) = ((\texttt{x}_1,\ldots,\texttt{x}_k),\texttt{e}_0) \\ [\texttt{d}'_1/\texttt{x}_1,\ldots,\texttt{d}'_k/\texttt{x}_k,\mathsf{object}_{\texttt{C}}(\texttt{v}_1,\ldots,\texttt{v}_n)/\texttt{this}]\texttt{e}_0 \longrightarrow \texttt{v}'\end{array}}{\texttt{e.m(}\texttt{d}_1,\ldots,\texttt{d}_k\texttt{)} \longrightarrow \texttt{v}'} \qquad \text{(R-Invk)}$$

$$\frac{\texttt{e} \longrightarrow \mathsf{object}_{\texttt{C}}(\texttt{v}_1,\ldots,\texttt{v}_n) \qquad \texttt{C} <: \texttt{D}}{\texttt{(D)e} \longrightarrow \mathsf{object}_{\texttt{C}}(\texttt{v}_1,\ldots,\texttt{v}_n)} \qquad \text{(R-Cast)}$$

$$\frac{\texttt{e}_0 \longrightarrow \texttt{true} \qquad \texttt{e}_1 \longrightarrow \texttt{v}'}{(\texttt{e}_0\texttt{?}\texttt{e}_1\texttt{:}\ \texttt{e}_2) \longrightarrow \texttt{v}'} \qquad \text{(R-Cond-T)}$$

$$\frac{\texttt{e}_0 \longrightarrow \texttt{false} \qquad \texttt{e}_2 \longrightarrow \texttt{v}'}{(\texttt{e}_0\texttt{?}\texttt{e}_1\texttt{:}\ \texttt{e}_2) \longrightarrow \texttt{v}'} \qquad \text{(R-Cond-F)}$$

$$\frac{\texttt{e}_0 \longrightarrow \texttt{e}'_0 \qquad \texttt{e}_1 \longrightarrow \texttt{e}'_1 \qquad \Delta_{\texttt{OP}}(\texttt{e}'_0,\texttt{e}'_1) = \texttt{v}}{\texttt{e}_0\ \texttt{OP}\ \texttt{e}_1 \longrightarrow \texttt{v}} \qquad \text{(R-Op)}$$

Figure 8.5: EFJ computation

variables to types. The typing judgment for expressions has the form $\Gamma \vdash \texttt{e} : \texttt{T}$, meaning "in the environment Γ, expression $\texttt{e}$ has type $\texttt{T}$." The typing rules are syntax directed, with one rule for each form of expression, except for binary operators which have three rules.

Integer and boolean constants have straightforward types (rules T-Bool and T-Int); the type of a variable is given by the environment Γ (rule T-Var). An operator maps base type values to a base type value (rules T-OP-I-I, T-OP-I-B, and T-OP-B-B). A conditional takes a boolean test and two expressions of the same type. The type of a field lookup is given by the type of the object and its class definition (rule T-Field). In a method invocation, the type of each argument must be a subtype of the type of the corresponding parameter, and the type is the return type of the method (rule T-Invk). In an object instantiation, the constructor arguments must match the fields of the object, and the type is that of the class that is instantiated (rule T-New). A typecast must either be downwards or upwards in the class hierarchy (rule T-Cast).

A method is well typed if the type of its body corresponds to the type signature of the method. A class is well typed if its constructor correctly passes arguments to the superclass, and each method is well typed. A program is well typed in an environment Γ_0 that binds any free variables in the main expression if all classes are well typed and the main expression is well typed in Γ_0.

8.2.4 EFJ reduction

We define EFJ computation using the eager big-step semantics shown in Figure 8.5. The reduction rules define evaluation of an expression into a value, as

$$
\begin{array}{lll}
\mathtt{2P} & ::= & \mathtt{(\{2CL_1,\ldots,2CL_n\},2e)} \\
\mathtt{2CL} & ::= & \mathtt{class\ C\ extends\ C\ \{T_1\ f_1;\ldots;T_n\ f_n;\ 2K\ 2M_1\ldots 2M_k\}} \\
\mathtt{2K} & ::= & \mathtt{K} \mid \underline{\mathtt{K}} \\
\mathtt{K} & ::= & \mathtt{C(T_1\ f_1,\ldots,T_n\ f_n)} \\
 & & \ \mathtt{\{super(f_i,\ldots,f_j);\ this.f_1\ =\ f_1;\ldots;this.f_n\ =\ f_n\}} \\
\mathtt{2M} & ::= & \mathtt{T\ m(2D_1,\ldots,2D_n)\ \{return\ 2e;\}} \\
 & \mid & \mathtt{T\ m(2D_1,\ldots,2D_n)\ \{}\underline{\mathtt{return}}\mathtt{\ 2e;\}} \\
\mathtt{2D} & ::= & \mathtt{T\ x} \mid \underline{\mathtt{T\ x}} \\
\mathtt{2e} & ::= & \mathtt{e} \mid \mathtt{lift(e)} \\
 & \mid & \underline{\mathtt{x}} \mid \mathtt{2e}\underline{\mathtt{.f}}_{\{\mathtt{C}_1,\ldots,\mathtt{C}_k\}} \mid \mathtt{2e}\underline{\mathtt{.m}}_{\{\mathtt{C}_1,\ldots,\mathtt{C}_k\}}\mathtt{(2e_1,\ldots,2e_n)} \\
 & \mid & \underline{\mathtt{new\ C}}\mathtt{(2e_1,\ldots,2e_n)} \mid \underline{\mathtt{(C)}}\mathtt{2e} \mid \mathtt{2e}\ \underline{\mathtt{OP}}\ \mathtt{2e} \mid \mathtt{(2e}\underline{\mathtt{?}}\mathtt{2e}\underline{\mathtt{:}}\mathtt{2e)} \\
\mathtt{e} & ::= & \mathtt{c} \mid \mathtt{x} \mid \mathtt{e.f}_{\{\mathtt{C}_1,\ldots,\mathtt{C}_k\}} \mid \mathtt{e.m}_{\{\mathtt{C}_1,\ldots,\mathtt{C}_k\}}\mathtt{(e_1,\ldots,e_n)} \\
 & \mid & \mathtt{new\ C(e_1,\ldots,e_n)} \mid \mathtt{(C)e} \mid \mathtt{e\ OP\ e} \mid \mathtt{(e?e:e)} \\
\mathtt{OP} & ::= & \mathtt{+} \mid \mathtt{-} \mid \mathtt{*} \mid \mathtt{/} \mid < \mid > \mid \mathtt{==} \mid \mathtt{\&\&} \mid \mathtt{||} \\
\mathtt{c} & ::= & \mathtt{true} \mid \mathtt{false} \mid \mathtt{0} \mid \mathtt{1} \mid \mathtt{-1} \mid \ldots \\
\multicolumn{3}{l}{\text{Values that result from computation:}} \\
\mathtt{v} & ::= & \mathtt{c} \mid \mathsf{object}_{\mathsf{C}}\mathtt{(v_1,\ldots,v_n)} \mid \mathsf{residual\ program\ part}
\end{array}
$$

Figure 8.6: 2EFJ syntax

follows. A **new** expression creates an object holding the value of each expression passed to the constructor (rule R-New); the constructor always initializes the fields of the object with the values of its arguments, as expressed directly in the semantics. A reference to a field retrieves the corresponding value (rule R-Field). Method invocation first reduces the self expression to decide the class of the receiver object, which determines what method is called; the method body is reduced after substitution of the values of the arguments for the formal parameters (rule R-Invk). Class casts can be reduced when the class of the concrete object is a sub-class of the casted type (rule R-Cast); if the concrete object is not a sub-class, the expression cannot be reduced. A conditional evaluates the test and only the appropriate branch (rules R-Cond-T and R-Cond-F). All operators are strict (rule R-Op); the operators are defined using an auxiliary function Δ, given in Figure A.6 of the appendix.

To compute the value of a complete program, the main expression of the program must be evaluated in an environment that defines the values of any free variables in the main expression. The typing rules of Figure 8.4 ensure that evaluation of a well-typed program according to the reduction rules of Figure 8.5 continues until a value has been computed, stops at an illegal type cast, or diverges.

8.3 Two-level Language

Partial evaluation can be formalized as execution in a language with a two-level syntax. The two-level separation of a program corresponds to the separation of a program into static and dynamic parts; we refer to the two levels as the static and the dynamic language levels. During evaluation of a two-level program, expressions in the static language level are reduced away, and expressions in the dynamic language level are residualized.

We extend EFJ into a two-level language by adding dynamic counterparts to most parts of the EFJ syntax, as shown in Figure 8.6; we name this language Two-Level EFJ (2EFJ). Static 2EFJ constructs are written as their EFJ counterparts, whereas dynamic constructs are underlined. We use monovariant binding times, which means that there is exactly one binding time associated with each program point.

We assign the same binding time to all objects of the same class, which we indicate by a binding-time annotation on the constructor of the class. We refer to a class with a statically-annotated constructor as a static class, and similarly for dynamic annotations. Intuitively, when a class is dynamic, all field references and method invocations on objects of that class are considered dynamic and will be residualized. Conversely, when a class is static, all field references and method invocations on objects of that class are removed by specialization. For method invocations, the binding-time annotation on the class indicates the binding time of the self; the binding time of each parameter other than the self depends on the binding times of the arguments that the method may be passed. The annotation on the return keyword of a method indicates the binding time of the method return value. An expression can be a static expression in standard EFJ syntax, a lift, or a dynamic expression. The lift operator allows a static, non-object value to appear in a dynamic context. Apart from constants (which are always static), each standard expression has a dynamic counterpart. The domain of values computed by the program is extended to not only include booleans, integers and objects, but also residual program parts.

To simplify partial evaluation for our language, we annotate field accesses and method invocation expressions with the list of possible classes of the receiver object; the binding time of a field access or method invocation expression is determined using the binding times of the classes included in the type annotation. The type annotations can trivially be computed using the type inference rules of Figure 8.4: for a given inferred type, an expression is annotated with the complete set of possible subtypes. More precise type-annotations (i.e., including a smaller set of types) can be obtained in many cases by using a more precise type inference algorithm, several of which are presented in literature [124, 125, 130]. More precise type-annotations can improve the precision of the binding-time analysis by allowing more of the program to be annotated as static, since expressions with disjoint type annotations can be given binding times independently.

8.4 Well-Annotatedness

Specialization of an EFJ program is done in terms of reduction of a binding-time annotated 2EFJ program. The 2EFJ counterpart to EFJ typing rules is referred to as *well-annotatedness:* a well-annotated (and well-typed) 2EFJ program either diverges, stops at the reduction of an illegal type cast, or reduces to a specialized program. Indeed, whether a program is well-annotated can be expressed as a type-checking problem with binding times as types.

In this section, we present the binding times that we use to describe well-annotatedness, consider what it means for a program to be well-annotated, and last define a type system that defines well-annotatedness for 2EFJ programs. We do not describe how to infer binding times for a program; this is the subject of Section 8.6.

8.4.1 Binding times

We use three different binding times to annotate programs: D, S_L, and S_N. D indicates a dynamic binding time. S_L indicates a static binding time on an expression that produces a value that can be lifted using a `lift` operator to produce a residual representation of the value (see next paragraph). S_N indicates a static binding time on an expression that produces a value that cannot be lifted. We write S_γ for a binding time that may be either S_L or S_N.

We consider integers and booleans to be the only liftable values types. We do not allow objects to be lifted; object lifting would have to be done by generating `new` expressions, which would duplicate computation if the same object were residualized in many places. Besides being inefficient, creating duplicate objects would cause inconsistency problems in a language where object references can be compared.

8.4.2 Considerations

The binding-time annotations to compute are monovariant; only one annotation is permitted per program point. In addition, all instances of the same class share the same binding time, given by the annotation on the constructor. For simplicity, we let all fields that belong to the same class have the binding time of the class, and we let binding times be flow-insensitive. (With flow-insensitive binding times, if at some program point a dynamic instance of a class is created, all instances of this class throughout the program will be considered dynamic.)

The binding time of two objects that are used at the same program point (through field lookup or method invocation) must be equal. Fields common to these two objects must have the same binding times, and all possible receiver methods must have equivalent binding times for the self and any other arguments. As explained earlier, we use the type inference system of Section 8.2 to derive expression type annotations; for a given field lookup or method invocation with a qualifying type T, this type system will derive the entire set of sub-types of T as possible types. Thus, with the exception of the special class `Object`, a class will have the same binding time as its superclass. The class `Object` is implicitly exempted since it has neither fields nor methods, and thus never can occur in an expression type annotation

Within these restrictions, we can still specialize small examples in interesting ways; for a discussion of what is required from a binding-time analysis to treat realistic applications, see the next chapter.

8.4.3 Well-annotatedness rules

We define well-annotatedness of a 2EFJ program using the type checking rules of Figure 8.7. These rules ensure that during evaluation of a 2EFJ program,

Expressions:

$$\tau \vdash \mathtt{c} : S_L \quad (\text{W-Const}) \qquad \tau \vdash \mathtt{x} : \tau(\mathtt{x}) \quad (\text{W-Var})$$

$$\frac{\tau \vdash \mathtt{e} : S_L}{\tau \vdash \mathtt{lift(e)} : D} \quad (\text{W-Lift})$$

$$\frac{\begin{array}{c}\tau \vdash \mathtt{e} : S_N \\ \textit{field-bt}(\mathtt{C_i}, \mathtt{f}) = S_\gamma\end{array}}{\tau \vdash \mathtt{e.f}_{\{\mathtt{C_1},\ldots,\mathtt{C}_k\}} : S_\gamma} \quad (\text{W-S-Field}) \qquad \frac{\begin{array}{c}\tau \vdash \mathtt{e} : D \\ \textit{field-bt}(\mathtt{C_i}, \mathtt{f}) = D\end{array}}{\tau \vdash \mathtt{e}\underline{\mathtt{.f}}_{\{\mathtt{C_1},\ldots,\mathtt{C}_k\}} : D} \quad (\text{W-D-Field})$$

$$\frac{\begin{array}{c}\tau \vdash \mathtt{e_i} : S_\gamma \\ \textit{class-bt}(\mathtt{C}) = S_N\end{array}}{\tau \vdash \mathtt{new\ C(e_1,\ldots,e_n)} : S_N} \quad (\text{W-S-New}) \qquad \frac{\begin{array}{c}\tau \vdash \mathtt{e_i} : D \\ \textit{class-bt}(\mathtt{C}) = D\end{array}}{\tau \vdash \underline{\mathtt{new\ C}}\mathtt{(e_1,\ldots,e_n)} : D} \quad (\text{W-D-New})$$

$$\frac{\tau \vdash \mathtt{e} : S_N}{\tau \vdash \mathtt{(C)e} : S_N} \quad (\text{W-S-Cast}) \qquad \frac{\tau \vdash \mathtt{e} : D}{\tau \vdash \underline{\mathtt{(C)}}\mathtt{e} : D} \quad (\text{W-D-Cast})$$

$$\frac{\tau \vdash \mathtt{e_1}, \mathtt{e_2} : S_L}{\tau \vdash \mathtt{e_1\ OP\ e_2} : S_L} \quad (\text{W-S-OP}) \qquad \frac{\tau \vdash \mathtt{e_1}, \mathtt{e_2} : D}{\tau \vdash \mathtt{e_1}\ \underline{\mathtt{OP}}\ \mathtt{e_2} : D} \quad (\text{W-D-OP})$$

$$\frac{\begin{array}{c}\tau \vdash \mathtt{e_0} : S_L \\ \tau \vdash \mathtt{e_1}, \mathtt{e_2} : T\end{array}}{\tau \vdash \mathtt{(e_0?e_1:e_2)} : T} \quad (\text{W-S-Cond}) \qquad \frac{\begin{array}{c}\tau \vdash \mathtt{e_0} : D \\ \tau \vdash \mathtt{e_1}, \mathtt{e_2} : D\end{array}}{\tau \vdash \mathtt{(e_0}\underline{\mathtt{?}}\mathtt{e_1}\underline{\mathtt{:}}\mathtt{e_2)} : D} \quad (\text{W-D-Cond})$$

$$\frac{\tau \vdash \mathtt{e} : S_N \quad \tau \vdash \mathtt{e_i} : T_i \quad \textit{bt-signature}(\mathtt{C_j}, \mathtt{m}) = S_N.(T_1,\ldots,T_n) \mapsto T_R}{\tau \vdash \mathtt{e.m}_{\{\mathtt{C_1},\ldots,\mathtt{C_k}\}}\mathtt{(e_1,\ldots,e_n)} : T_R} \quad (\text{W-S-Invk})$$

$$\frac{\tau \vdash \mathtt{e} : D \quad \tau \vdash \mathtt{e_i} : T_i \quad \textit{bt-signature}(\mathtt{C_j}, \mathtt{m}) = D.(T_1,\ldots,T_n) \mapsto D}{\tau \vdash \mathtt{e}\underline{\mathtt{.m}}_{\{\mathtt{C_1},\ldots,\mathtt{C_k}\}}\mathtt{(e_1,\ldots,e_n)} : D} \quad (\text{W-D-Invk})$$

Methods:

$$\frac{\begin{array}{c}\textit{bt-signature}(\mathtt{C}, \mathtt{m}) = T_0.(T_1,\ldots,T_n) \mapsto S_\gamma \\ \tau = \textit{build-env}(\textit{no-bt}(P), T_0, (T_1,\ldots,T_n)) \qquad \tau \vdash \mathtt{e} = S_\gamma\end{array}}{\mathtt{T\ m}(P)\ \{\ \mathtt{return}\ e;\ \}\ \mathtt{WF\ IN\ C}} \quad (\text{W-S-Method})$$

$$\frac{\begin{array}{c}\textit{bt-signature}(\mathtt{C}, \mathtt{m}) = T_0.(T_1,\ldots,T_n) \mapsto D \\ \tau = \textit{build-env}(\textit{no-bt}(P), T_0, (T_1,\ldots,T_n)) \qquad \tau \vdash \mathtt{e} = D\end{array}}{\mathtt{T\ m}(P)\ \{\ \underline{\mathtt{return}}\ e;\ \}\ \mathtt{WF\ IN\ C}} \quad (\text{W-D-Method})$$

Classes:

$$\frac{k \in 2\mathtt{K} \qquad \mathtt{M}_i\ \mathtt{WF\ IN\ C}}{\mathtt{class\ C\ extends\ D}\ \{\ \mathtt{C_1\ f_1};\ldots;\mathtt{C}_n\ \mathtt{f}_n\ k\ \mathtt{M_1}\ \ldots\ \mathtt{M}_p\ \}\ \mathtt{WF}}$$

Program:

$$\frac{\forall \mathtt{CL} \in \mathtt{CL_1},\ldots,\mathtt{CL}_n : \mathtt{CL\ WF} \qquad \tau_0 \vdash \mathtt{e} \in \mathtt{T}}{(\mathtt{CL_1},\ldots,\mathtt{CL}_n, \mathtt{e})\ \mathtt{WF\ IN}\ \tau_0}$$

Binding times BT: S_L, S_N, D, with $S_L \sqsubset D$ and $S_N \sqsubset D$ (notation: $S_\gamma \in \{S_L, S_D\}$)
Binding-time environment $\tau : \mathsf{Var} \to \mathsf{BT}$

Figure 8.7: Rules for well-annotatedness

no residual program parts appear where a value is expected, and that liftable values (i.e., objects) are not required to be residualized into the program. In these rules, an environment τ is a finite mapping from variables to binding times. The well-annotatedness judgment for expressions has the form $\tau \vdash \mathtt{e} : T$, meaning "in the environment τ, expression e has binding time T." The well-annotatedness rules are syntax directed, with two rules for each kind of 2EFJ expression, except constants, variables, and the lift operator, which each only have a single rule.

A number of auxiliary definitions are used in Figure 8.7; we summarize them here, and refer to Figure A.3 of the appendix for details. The function *class-bt* returns the binding time of a class. The function *field-bt* returns the binding time of a field; when a class is static, the binding-time of a field depends on the type of the field. The function *bt-signature* determines the binding-time signature of a method based on the binding-time annotations in the class that defines the method (namely the annotations on the class, the formal parameters, and return statement of the method). The function *no-bt* maps a 2EFJ program part into an EFJ program part by removing binding-time annotations and uses of the lift operator. The function *build-env* builds a binding-time environment from a list of formal parameters and a list of binding times to associate with these parameters.

The rules of Figure 8.7 are defined as follows. Constants must always be annotated S_L (rule W-CONST); the binding-time annotation of a variable is given by the environment τ (rule W-VAR); the lift operator makes a residualizable static binding time be dynamic (rule W-LIFT). The binding-time annotation of a field access must correspond to the binding time of the field, across all classes that may be used at the program point (rules W-S-FIELD and W-D-FIELD). The binding time of a field is equivalent to the binding time of the class that contains the field. The binding time of an object instantiation must be equivalent to the binding time of the class that is being instantiated (rules W-S-NEW and W-D-NEW). Similarly, the binding time of a cast must be equivalent to the binding time of the class that it is being cast to (rules W-S-CAST and W-D-CAST). The binding time of an operator must correspond to the binding time of its arguments (rules W-S-OP and W-D-OP). The binding time of a conditional corresponds to the binding time of its branches, and the annotation on the conditional itself must correspond to the binding time of the test. Furthermore, when the test is dynamic, the branches must be dynamic as well (rules W-S-COND and W-D-COND).

For a method invocation, the binding time of the self object must correspond to the binding time of the classes of the possible receiver objects, and the binding times of the parameters must correspond to the binding times of the actual arguments. The auxiliary function *bt-signature* is used to look up the binding-time signature of the method.

For the binding-time annotations of a method declaration to be well-formed, the binding time of its body must correspond to the binding-time annotation on the `return` statement. This is checked using the well-annotatedness rules for expressions, in an environment defined by the binding-time annotations on the class and the method formal parameters. The environment is created using the auxiliary function *build-env* (which takes the names of the formal parameters as its first argument). For the binding-time annotations of a class to be well-formed, the binding-time annotation of each method must be well-formed. Similarly, for the binding-time annotations of a program to be well-formed in an environment τ_0 that provides binding times for any free variables, the binding-time annotation of each class must be well-formed.

8.5 Specialization

For a given 2EFJ program, the static parts can be reduced away, leaving behind only the dynamic parts. Reduction of a static part of the program is done using the standard reduction rules of EFJ. However, the reduction of a static program part (e.g., a conditional) can result in a residual program part, so residual program parts are considered an additional form of value. Reduction of a dynamic program construct yields a residual program part. We consider reduction of well-annotated 2EFJ programs, so reduction either diverges, stops at an illegal static type cast, or results in a specialized program.

In this section, we define 2EFJ reduction rules that define specialization of EFJ programs. First, we describe the structure of the 2EFJ reduction rules. Then, we define reduction of static program parts, and subsequently reduction of dynamic program parts. Last, we define reduction of a complete program, and present a number of improvements to the standard reduction rules.

8.5.1 2EFJ expression reduction

Figure 8.8 shows the definition of 2EFJ reduction. The reduction rules evaluate an expression into a tuple; the first member is a value (possibly in the form of a residual expression), and the second member is a set of fresh method definitions. Partial evaluation of an object-oriented program introduces new methods into classes, which is expressed by the 2EFJ reduction rules. The second member of the pair that results from evaluation of an expression holds the set of methods that are to be introduced into the program as a result of specializing this expression.

8.5.2 Reduction of static expressions

The static parts of a 2EFJ expression reduce into values using a set of rules that are counterparts to the standard EFJ reduction rules of Figure 8.5, extended to collect specialized methods. Reduction of a static expression never produces residual methods to be introduced into the classes of the program, so the extended EFJ rules simply thread the set of new methods through the evaluation of each sub-expression. The 2EFJ counterpart of an EFJ reduction rule R-X is named 2-R-S-X (**r**educe **s**tatic).

The reduction rules for static 2EFJ expressions are as follows. The parameters to a static object constructor are assumed to be always static, so static object instantiation creates a standard object (rule 2-R-S-NEW). A field access is performed as usual when the self object is static (rule 2-R-S-FIELD). A method

Each EFJ reduction rule from Figure 8.5 is extended to collect residual specialized methods in the same way as the 2EFJ reduction rules below. The 2EFJ version of an EFJ rule (R-X) is named (2-R-S-X), giving the rules (2-R-S-New), (2-R-S-Field), (2-R-S-Invk), (2-R-S-Cast), (2-R-S-Cond), and (2-R-S-OP).

$$\frac{e \longrightarrow (r, M)}{\mathtt{lift}(e) \longrightarrow (\mathit{build\text{-}const}(r), M)} \qquad \text{(2-R-Lift)}$$

$$\frac{\mathrm{vn} = \mathit{name}(\mathtt{x})}{\underline{\mathtt{x}} \longrightarrow (\mathit{build\text{-}var}(\mathrm{vn}), \emptyset)} \qquad \text{(2-R-D-Var)}$$

$$\frac{e_i \longrightarrow (r_i, M_i) \qquad \mathrm{cn} = \mathit{name}(\mathtt{C})}{\underline{\mathtt{new\ C}}(e_1, \ldots, e_n) \longrightarrow (\mathit{build\text{-}new}(\mathrm{cn}, (r_1, \ldots, r_n)), \cup M_i)} \qquad \text{(2-R-D-New)}$$

$$\frac{e \longrightarrow (r, M) \qquad \mathrm{fn} = \mathit{name}(\mathtt{f})}{e\underline{.\mathtt{f}}_{\{\mathtt{C}_1,\ldots,\mathtt{C}_k\}} \longrightarrow (\mathit{build\text{-}field\text{-}lookup}(\mathrm{fn}, r), M)} \qquad \text{(2-R-D-Field)}$$

$$\frac{\begin{array}{c} e_i \longrightarrow (r_i, M_i) \\ \alpha_i = \mathit{build\text{-}subst}(\mathtt{C}_i, \mathtt{m}, (r_1, \ldots, r_n)) \quad \mathit{mbody}(\mathtt{C_i}, \mathtt{m}) = ((\ldots), d_i) \quad \alpha_i d_i \longrightarrow (d'_i, M'_i) \\ N = (\cup M_i) \cup (\cup M'_i) \quad \mathrm{mn} = \mathit{new\text{-}name}(N, \mathtt{m}) \quad m_i = \mathit{build\text{-}method}(\mathtt{C}_i, \mathtt{m}, \mathrm{mn}, d'_i) \end{array}}{e_0\underline{.\mathtt{m}}_{\{\mathtt{C}_1,\ldots,\mathtt{C}_k\}}(e_1, \ldots, e_n) \longrightarrow (\mathit{build\text{-}invoke}(\mathtt{m}, \mathrm{mn}, r_0, (r_1, \ldots, r_n)), N \cup \{m_1, \ldots, m_k\})}$$
(2-R-D-Invk)

$$\frac{e \longrightarrow (r, M) \qquad \mathrm{cn} = \mathit{name}(\mathtt{C})}{\underline{(\mathtt{C})}e \longrightarrow (\mathit{build\text{-}cast}(\mathrm{cn}, r), M)} \qquad \text{(2-R-D-Cast)}$$

$$\frac{e_i \longrightarrow (r_i, M_i)}{e_0\underline{?}e_1\underline{:}e_2 \longrightarrow (\mathit{build\text{-}if}(r_0, r_1, r_2), \cup M_i)} \qquad \text{(2-R-D-Cond)}$$

$$\frac{e_i \longrightarrow (r_i, M_i) \qquad \mathrm{on} = \mathit{name}(\mathtt{OP})}{e_0\ \underline{\mathtt{OP}}\ e_1 \longrightarrow (\mathit{build\text{-}op}(\mathrm{on}, r_0, r_1), \cup M_i)} \qquad \text{(2-R-D-OP)}$$

Residual methods produced M: $\{(\mathsf{Class}, \mathsf{Type}, \mathsf{Method}, (\mathsf{Var} \times \ldots \times \mathsf{Var}), \mathsf{Exp})\}$
build-X$(v_1, \ldots, v_n)$=residual form X with subcomponents $v_1, \ldots, v_n$
name(`x`)=residual representation of variable `x`
new-name$(M, \mathtt{m})$=new method name based on `m` but not defined in M or the program

Figure 8.8: Specialization as two-level execution

invocation with a static self object but a dynamic return value will produce a residual expression that is unfolded into the calling context; any arguments, be they values or residual program parts, are substituted throughout the body of the method (rule 2-R-S-Invk). A static type cast is assumed to always be given a static object, and therefore reduces in the standard way (rule 2-R-S-Cast). A conditional with a static test and dynamic branches will reduce to a residual expression (rule 2-R-S-Cond). A static operator is assumed to only have static arguments, and therefore reduces in the standard way (rule 2-R-S-OP).

The reduction of class casts would seem to cause problems for the well-typedness of the residual program. However, a class cast is only reduced away when its object argument is completely static, in which case no operations performed on this object are residualized into the program. The case is similar for the implicit cast of the self object that occurs in a virtual dispatch: unfolding

$$\frac{\begin{array}{c} \mathit{mbody}(\mathtt{C},\mathtt{m}) = ((\mathtt{x_1},\ldots,\mathtt{x_n}), e) \\ \mathit{bt\text{-}signature}(\mathtt{C},\mathtt{m}) = D.(T_1,\ldots,T_n) \mapsto D \qquad \beta_i = \mathit{mk\text{-}subst}(T_i, \mathtt{x}_i, r_i) \end{array}}{\mathit{build\text{-}subst}(\mathtt{C},\mathtt{m},(r_1,\ldots,r_n)) = \beta_1 \ldots \beta_n}$$

$$\mathit{mk\text{-}subst}(S_\gamma, \mathtt{x}, r) = [r/\mathtt{x}] \qquad \mathit{mk\text{-}subst}(D, \mathtt{x}, r) = []$$

$\mathit{build\text{-}method}(\mathtt{C},\mathtt{m},\mathrm{mn},d)$ =new method "mn" in class C with body d,
with the dynamic formal parameters of m.

Figure 8.9: Auxiliary definitions for Figure 8.8

only takes place when the self object is static and hence is not residualized.

8.5.3 Reduction of dynamic expressions

Reduction of a use of the lift operator residualizes a value resulting from a static computation into the program(2-R-Lift). With the exception of method invocation, all other reduction rules for dynamic constructs are straightforward: each sub-component is reduced into a residual expression, and used to rebuild the construct (rules 2-R-D-Var, 2-R-D-New, 2-R-D-Field, 2-R-D-Cast, 2-R-D-Cond, and 2-R-D-OP).

To reduce a dynamic method invocation, a new set of virtual methods must be generated, one for each possible receiver class (rule 2-R-D-Invk). To this end, the arguments and the self object of the method invocation are first reduced; then, each possible callee is specialized. The specialized body of each callee is obtained by reduction after substitution of its static parameters throughout the body of the generic method. Each resulting body is inserted into a fresh method which has the same name across all of the possible receiver classes, and which is added to the set of produced methods. The reduction rule for method invocation uses two auxiliary definitions which are shown in Figure 8.9.

In the 2EFJ reduction rule for dynamic method invocation, we exploit the fact that our type inference algorithm always includes the class of the qualifying type in the set of classes used to annotate virtual dispatches. Thus, a specialized method is always generated for the class of the qualifying type. If this were not the case, the residualized virtual dispatch might be illegal: there could be a residual virtual dispatch to a virtual method not defined in the class of the qualifying type. To circumvent the problem when using a more precise type inference algorithm, a dummy method could be introduced into the class of the qualifying type, to ensure a correct program.

8.5.4 Reduction of a program

Reduction of a 2EFJ program produces a specialized main expression and a collection of specialized methods; this representation can be transformed into the aspect syntax of Figure 8.10a. A program in our formalized EFJ language has a main expression which is specialized by our partial evaluator, so the aspect syntax must support expressing a specialized main expression. We use `introduction` blocks to introduce specialized methods into classes, and a special

```
ASPECT          ::=  aspect {
                       (INTRODUCTION)*
                       main EFJ-EXPRESSION
                     }
INTRODUCTION    ::=  introduction CLASS-NAME {
                       (EFJ-METHOD)*
                     }
CLASS-NAME      ::=   ...(EFJ class name)...
```

(a) Aspect syntax for program with main expression

$$\frac{\begin{array}{c} e \longrightarrow (M, e') \\ \{\mathtt{M_1},\dots,\mathtt{M_n}\} = \{\mathtt{T\ m\ (x_1,\dots,x_k)\{\ return}\ e;\}|(\mathtt{C}_i,\mathtt{T},\mathtt{m},(\mathtt{x_1},\dots,\mathtt{x_k}),e) \in M\} \\ I_i = \mathtt{introduction\ C}_i\ \{\ \mathtt{M_1},\dots,\mathtt{M_n}\ \} \end{array}}{(\{\mathtt{C_1},\dots,\mathtt{C_n}\}, e) \longrightarrow \mathtt{aspect}\ \{\mathtt{I_1}\dots\mathtt{I}_n\ \mathtt{main}\ e'\}}$$

(b) Specialization into an aspect

$$\frac{\begin{array}{c} I_i = \mathtt{introduction\ C}_i\ \{\ \mathtt{M'_1},\dots,\mathtt{M'_k}\ \} \qquad \mathtt{C_i} = \mathtt{class\ C\ extends\ D}\ \{\dots;\mathtt{K\ M_1}\dots\mathtt{M_n}\} \\ \mathtt{C'_i} = \mathtt{class\ C\ extends\ D}\ \{\dots;\mathtt{K\ M_1}\dots\mathtt{M_n\ M'_1}\dots\mathtt{M'_k}\} \end{array}}{\mathit{weave}((\{\mathtt{C_1},\dots,\mathtt{C_n}\},e),\mathtt{aspect}\ \{\ \mathtt{I_1},\dots,\mathtt{I}_n\ \ \mathtt{main}\ e'\ \}) \longrightarrow (\{\mathtt{C'_1},\dots,\mathtt{C'_n}\},e')}$$

(c) Weaving of aspect and program

Figure 8.10: Specialization of a program into an aspect

`main` block to replace the main expression of a program. The rules for transforming a reduced 2EFJ expression into an aspect are shown in Figure 8.10b. The aspect produced by specialization can be woven into the main program using a simple weaver *weave*, defined by the reduction rule of Figure 8.10c. The overall effect is that each specialized method is inserted into the class for which it was specialized, and that the generic main expression is replaced by the specialized main expression.

8.5.5 Improved 2EFJ expression reduction

The 2EFJ reduction rules for method invocation shown in Figure 8.8 suffer from two major problems. First, the rule for reducing a static method invocation can cause code duplication. Second, the rule for reducing a dynamic method invocation cannot reduce a method that performs recursive calls under dynamic control. Both problems are well-known in the field of partial evaluation, and are easy to solve.

A dynamic residual expression passed to a static method invocation will be duplicated inside the body of the method if the corresponding formal parameter of the method appears many times. The standard solution is to use a let expression (or some other form of local variable) to bind such a dynamic expression to a variable, and then substitute this variable throughout the body of the method. The only form of local variables in EFJ are method parameters, so we would need to use a wrapper method to perform a local binding. However, using a wrapper would introduce a new virtual dispatch into the program, so it would be preferable to add a let expression to EFJ. The treatment of let expressions is well known in partial evaluation literature, so we will not consider this problem any further.

With the reduction rule of Figure 8.8, it is not possible to reduce a dynamically controlled recursive method: reduction of such a method would give rise to an infinitely large reduction tree, i.e., specialization would not terminate. The rule for reducing a dynamic method invocation does not check first whether a usable specialized method exists before generating a new one in a depth-first manner. The solution is to introduce a cache of specialized methods, which will let us re-use methods that have been generated for a specific combination of possible callee classes and static values. A list should be kept of which methods are currently being specialized for what values, to avoid infinite generation of identical methods. It is fairly straightforward to add a method cache to the rules of Figure 8.8, and the resulting rules are shown in the appendix in Figure A.4.

8.6 Binding-Time Analysis

Binding-time analysis of an EFJ program derives annotations to divide the program into static and dynamic 2EFJ program parts. The binding-time analysis is supplied the binding times of the free variables of the main expression, and the derived annotations must respect the well-annotatedness rules while making static as large a part of the program as is possible. We express the binding-time analysis as constraints on the binding times of the program, and then use a constraint solver to find a consistent solution that within these constraints retains as static the largest possible part of the program.

This section presents a contraint-based binding-time analysis for EFJ. First, we describe the notation that we use for writing constraints. Then, we present an algorithm for generating constraints for a given program, and last we show how these constraints can be solved to derive binding times for the program.

8.6.1 Constraint system

We generate one or more constraints for every program part that can be annotated. A constraint variable that is used to express constraints on the binding time of an entity e is written T_e. So, if e is a field access expression, then T_e is a variable used to constrain the binding time of this field access expression; additional constraints are generated for each sub-component of the field access expression. Classes, conditionals and methods are special cases. We use $T_{\mathtt{C}}$ to constrain the binding time of a class `C`. The binding time of the value produced by a conditional may be different from the annotation on the conditional; for a conditional expression e, we use T_e as a constraint on the value produced by e, and $T_{e?}$ as a constraint on the binding time of the conditional itself. For a method `m` in the class C with formal parameters $\mathtt{x}_1,\ldots,\mathtt{x}_n$, we use $T_{\mathtt{C.m.return}}$

$$
\begin{array}{lcl}
\multicolumn{3}{l}{\mathcal{C}^E(\mathtt{C},\mathtt{m},e) = \text{case } e \text{ of}} \\
\quad [\![\mathtt{c}]\!] & \Rightarrow & \{S_L = T_e\} \\
\quad [\![\mathtt{x}]\!] & \Rightarrow & \{T_{\mathtt{C.m.x}} = T_e\} \\
\quad [\![e_1.\mathtt{f}_{\{\mathtt{C_1},\ldots,\mathtt{C_k}\}}]\!] & \Rightarrow & \left(\begin{array}{l} \{T_{e_1} = T_e, T_{e_1} = T_{\mathtt{C}_i}\},\ \textit{object-type}(\{\mathtt{C}_1,\ldots,\mathtt{C}_k\},\mathtt{f}) \\ \{T_{e_1} \rhd T_e, T_{e_1} = T_{\mathtt{C}_i}, T_e \preceq D\},\ \text{otherwise} \end{array} \right. \\
 & & \cup\, \mathcal{C}^E(\mathtt{C},\mathtt{m},e_1) \\
\quad [\![\mathtt{new}\ \mathtt{C}(e_1,\ldots,e_n)]\!] & \Rightarrow & \left\{T_{e_i} \preceq \overline{T}_{e_i}, \overline{T}_{e_i} = T_{\mathtt{C}}, T_e = T_{\mathtt{C}}\right\} \cup \mathcal{C}^E(\mathtt{C},\mathtt{m},e_i) \\
\quad [\![(\mathtt{C})e_1]\!] & \Rightarrow & \{T_{e_1} = T_e\} \cup \mathcal{C}^E(\mathtt{C},\mathtt{m},e_1) \\
\quad [\![e_1\ \mathtt{OP}\ \mathtt{e_2}]\!] & \Rightarrow & \left\{T_{e_i} \preceq \overline{T}_{e_i}, \overline{T}_{e_i} = T_e\right\} \cup \mathcal{C}^E(\mathtt{C},\mathtt{m},e_1) \cup \mathcal{C}^E(\mathtt{C},\mathtt{m},e_2) \\
\quad [\![(e_1?e_2{:}e_3)]\!] & \Rightarrow & \left\{ \begin{array}{l} T_{e_2} \preceq \overline{T}_{e_2}, T_{e_3} \preceq \overline{T}_{e_3}, T_{e_1} \rhd \overline{T}_{e_2}, T_{e_1} \rhd \overline{T}_{e_3}, \\ T_{e_?} = T_{e_1}, \overline{T}_{e_2} = \overline{T}_{e_3}, T_e = \overline{T}_{e_2}, T_e = \overline{T}_{e_3} \end{array} \right\} \\
 & & \cup\, \mathcal{C}^E(\mathtt{C},\mathtt{m},e_1) \cup \mathcal{C}^E(\mathtt{C},\mathtt{m},e_2) \cup \mathcal{C}^E(\mathtt{C},\mathtt{m},e_3) \\
\quad [\![e_1.\mathtt{m}_{\{\mathtt{C_1},\ldots,\mathtt{C_k}\}}(d_1,\ldots,d_n)]\!] & \Rightarrow & \left\{T_{d_i} \preceq \overline{T}_{d_i}, \overline{T}_{d_i} = T_{\mathtt{C_j.m.x_i}}, T_{e_1} = T_{\mathtt{C_i}}, T_e \rhd T_{\mathtt{C_i.m.return}}\right\} \\
 & & \cup\, \mathcal{C}^E(\mathtt{C},\mathtt{m},e_1) \cup \mathcal{C}^E(\mathtt{C},\mathtt{m},d_i)
\end{array}
$$

$$\mathcal{C}^M(\mathtt{C},\mathtt{m},e) = \{T_e \preceq \overline{T}_e, T_{\mathtt{C.m.return}} = \overline{T}_e, T_{\mathtt{C}} \rhd T_{\mathtt{C.m.return}}\} \cup \mathcal{C}^E(\mathtt{C},\mathtt{m},e)$$

$$\mathcal{C}^C(\mathtt{class\ C\ extends\ D}\ \{\ldots;\mathtt{K}\ \mathtt{M_1}\ldots\mathtt{M_n}\}) = \cup\mathcal{C}^M(\mathtt{C},\mathtt{m}_i,e_i),\ \mathtt{M}_i = \mathtt{T}\ \mathtt{m}_i(\ldots)\ \{\ \mathtt{return}\ e_i;\ \}$$

$$\mathcal{C}^P(\{\mathtt{C_1},\ldots,\mathtt{C_n}\},e,I) = I \cup \mathcal{C}^E(\Box,\Box,e) \cup (\cup\mathcal{C}^C(\mathtt{C_i}))$$

Constraints: $S_L \prec D$

$T_1 \rhd T_2 \Leftrightarrow (T_1 = D \Rightarrow T_2 = D)$

Definitions: $\textit{object-type}(\{\mathtt{C}_1,\ldots,\mathtt{C}_k\},\mathtt{f}) = \forall\mathtt{C} \in \{\mathtt{C}_1,\ldots,\mathtt{C}_k\}:$

$\mathtt{T}\ \mathtt{f} \in \textit{fields}(\mathtt{C}), \mathtt{T} \notin \{\mathtt{int},\mathtt{boolean}\}$

Figure 8.11: Constraint generation

to constrain the binding time of the method return value, and $T_{\mathtt{C.m.x}_i}$ to constrain the binding time of each method parameter (the binding time of the self argument is constrained by $T_{\mathtt{C}}$).

Apart from equality between binding times, we use two constraint operators to express the EFJ binding-time analysis. The operator $\prec$ is used to constrain a liftable expression, and the operator $T_1 \rhd T_2$ expresses a dependency between T_1 and T_2: if T_1 is dynamic, then T_2 must also be dynamic. An expression e that may be liftable when it is of base type is represented by two binding-time variables, T_e and $\overline{T}_e$. The variable T_e represents the binding time of e, while $\overline{T}_e$ represents the binding time of the context of e. All constraints on uses of e are expressed relative to $\overline{T}_e$, and T_e is allowed to be static liftable while $\overline{T}_e$ is dynamic. For simplicity, expressions known not to be liftable (for example, the self object in a method invocation) are represented by a single binding time variable.

8.6.2 Constraint generation

The constraint system generator is shown in Figure 8.11, and is derived directly from the rules for well-annotatedness. The binding-time constraints for an expression `e` in a method `m` of the class `C` are generated using $\mathcal{C}^E(\mathtt{C},\mathtt{m},\mathtt{e})$. A constant should always be annotated as static liftable, and a local variable should have the same binding time throughout the method. For a field access, the constraint on the binding time depends on whether the field is of object type or base type; for an object type field, the binding time of the expression should be equal to the binding time of the self object, and the binding time of the self object should be equal to the binding times of the possible classes of the self; for a base type field, the binding time of the expression is dynamic when the self is dynamic, the binding time of the self should be equal to the binding times of the possible classes of the self, and the binding time of the expression should either be static liftable or dynamic. Object instantiation is similar to field access, except that constructor arguments might be liftable. The productions for class casts and operators enforce that the sub-components have the same binding times as the expression itself. For a conditional the binding times of the branches and of the value produced by the conditional depend on the binding time of the test; the binding time of the conditional itself depends only on the binding time of the test. Last, for a method invocation, the binding times of the arguments should correspond to the binding times of the formal parameters of each callee, and the binding time of the complete expression should correspond to the binding time of the return value of each callee.

To generate constraints for a program, constraints are generated for all methods of all classes, which ensures global consistency (the initial constraints on the free variables of the program are represented using the variable I). The special naming convention $T_{\Box.\Box.\mathtt{x}}$ is used to indicate a constraint on a free variable `x` of the main expression of the program. (This notation is reflected in the use of $\mathcal{C}^E(\Box, \Box, e)$ to generate constraints over the main expression.)

8.6.3 Constraint solving

To efficiently solve the constraint system generated for an EFJ program, we can directly use the constraint solver of the C-Mix partial evaluator for C [5, 6]. We only make use of three sorts of constraints $(=, \preceq, \triangleright)$, all of which are identical in C-Mix. We map our set of binding times $\{S_L, S_N, D\}$ into C-Mix binding times using a mapping α:

$$\begin{array}{lcl} \alpha(S_L) & = & S \\ \alpha(S_N) & = & *S \\ \alpha(D) & = & D \end{array}$$

In C-Mix, the binding time $*S$ indicates a pointer value, which is non-liftable in C. (We prefer the use of S_L and S_N for EFJ binding times, since $*$ is not used in EFJ in conjunction with non-liftable values.) The C-Mix constraint solver treats a richer set of binding times, but solving our constraint system does not generate new forms of binding times. Thus, to obtain the result of the binding-time analysis after having applied the C-Mix constraint solver, we can simply employ the obvious reverse mapping of α. The solution produced by the C-Mix constraint solver directly defines binding times for all dynamic program parts; static program parts are assigned binding times (static liftable or static non-liftable) based on their type. A constraint $T_e \prec \overline{T}_e$ in the solution indicates that a lift operator should be inserted around the expression e.

```
class Binary extends Object {
  Binary() { super(); }
  int eval( int x, int y ) {
    return this.eval( x, y );
  }
}
class Add extends Binary {
  Add() { super(); }
  int eval( int x, int y ) {
    return x+y;
  }
}
class Mult extends Binary {
  Mult() { super(); }
  int eval( int x, int y ) {
    return x*y;
  }
}
```

```
class Power extends Object {
  int exp; Binary op; int neutral;
  Power( int exp, Binary op, int neutral ) {
    super();
    this.exp = exp;
    this.op = op;
    this.neutral = neutral;
  }
  int raise( int base ) {
    return loop( base, exp );
  }
  int loop( int base, int e ) {
    return (e==0
     ? this.neutral
     : this.op.eval( base,
                     this.loop( base, e-1 ) ) );
  }
}
```

Figure 8.12: The power example (again)

8.7 Examples

In this section we present a few simple examples of how the partial evaluator defined in this chapter can be used to specialize object-oriented programs. First, we return to the power example of Chapter 5, and then we present a more complicated example involving a fold function over lists.

8.7.1 The power example

Figure 8.12 shows the Power class that was specialized in Chapter 5. The method `raise` is defined in terms of an auxiliary method `loop`, which calls itself recursively while applying the operator and decrementing the exponent value. We specialize this collection of classes in a program with main expression

```
(new Power(e,b,n)).raise(x)
```

and the types of the free variables given by the environment

$$\Gamma_1 = \{\texttt{e} \mapsto \texttt{int}, \texttt{b} \mapsto \texttt{Binary}, \texttt{n} \mapsto \texttt{int}, \texttt{x} \mapsto \texttt{int}\}$$

We consider `e`, `b`, and `n` to be known inputs.

We apply our binding-time analysis to this program, with an initial set of constraints

$$\tau_1 = \{T_{\Box,\Box,\texttt{b}} = S_L, T_{\Box,\Box,\texttt{e}} = S_N, T_{\Box,\Box,\texttt{n}} = S_L, T_{\Box,\Box,\texttt{x}} = D\}$$

which defines the binding times of the free variables of the main expression. The resulting binding-time annotation is illustrated in Figure 8.13, with underlining used to indicate dynamic binding times. The constraint that `x` is dynamic forces the formal parameter `base` of the methods `raise` and `loop` of the class `Power` to be dynamic. This constraint forces the formal parameters `x` and `y` of the `eval` methods of the `Binary`, `Add`, and `Mult` classes to be dynamic as well. All other program parts are not influenced by dynamic constraints and are annotated static.

```
class Binary extends Object {
  Binary() { super(); }
  int eval( int x, int y ) {
    return this.eval( x, y );
  }
}
class Add extends Binary {
  Add() { super(); }
  int eval( int x, int y ) {
    return x+y;
  }
}
class Mult extends Binary {
  Mult() { super(); }
  int eval( int x, int y ) {
    return x*y;
  }
}

   (new Power(e,b,n)).raise(x)
```

```
class Power extends Object {
  int exp; Binary op; int neutral;
  Power( int exp, Binary op, int neutral ) {
    super();
    this.exp = exp;
    this.op = op;
    this.neutral = neutral;
  }
  int raise( int base ) {
    return loop( base, exp );
  }
  int loop( int base, int e ) {
    return (e==0
     ? lift(this.neutral)
     : this.op.eval( base,
                     this.loop( base, e-1 ) ) );
  }
}
```

Figure 8.13: Binding-time annotated version of the power example

Given this binding-time division, we can specialize the program for a substitution that defines the values of the free values of the main expression. Concretely, we use the substitution

$$\rho_1 = [3/\texttt{e}, \texttt{new Mult()}/\texttt{b}, 1/\texttt{n}]$$

to specialize the program. No classes are annotated dynamic, so all field references and method invocations can be reduced during specialization. No specialized methods are generated, so the result is simply the specialized program

```
aspect { main x*x*x*1 }
```

The program could be further optimized using simple arithmetic properties, which would give `x*x*x` as a main as expression.

8.7.2 The fold example

In the power example, all classes are annotated completely static, and thus not referenced from the residual main expression. For a more interesting example, we use lists with a built-in fold function, as shown in Figure 8.14. The list fold function accepts a binary operator (the same as was used for the power example) and an initial value. As a main expression for the program, we use

```
ls.fold(op,init)
```

in a type environment defined by

$$\Gamma_2 = \{\texttt{ls} \mapsto \texttt{List}, \texttt{op} \mapsto \texttt{Binary}, \texttt{init} \mapsto \texttt{int}\}$$

The method `fold` folds an operator over a list with a given initial value; we specialize it in two different scenarios. First, we specialize for the list structure

```
class List extends Object {
  List() { super(); }
  int fold( Binary op, int init ) { return this.fold(op,init); }
}
class Empty extends List {
  Empty() { super(); }
  int fold( Binary op, int init ) { return init; }
}
class ListElm extends List {
  int i; List next;
  ListElm( int i, List next ) { this.i=i; this.next=next; }
  int fold( Binary op, int init ) {
    return op.eval(this.i,this.next.fold(op,init));
  }
}
```

Figure 8.14: Folding over lists

as static and the operator and initial value as dynamic. The result is that the operator `eval` methods are unfolded over the list elements. Then, we specialize for the opposite scenario, where the list is dynamic and the operator and initial value are static. Here, the result is a specialized method `fold` that directly uses the operator and neutral value. These scenarios are described in detail below.

Scenario 1: static list

To specialize for a static list argument, we provide the following initial constraint on the binding times of the free variables of the main expression:

$$\tau_2 = \{T_{\square,\square,\texttt{ls}} = S_N, T_{\square,\square,\texttt{op}} = D, T_{\square,\square,\texttt{init}} = D\}$$

We constrain the list argument to be static, and the binary operator and initial value arguments to be dynamic. The resulting binding-time annotation is illustrated in Figure 8.15. The constraint that `op` is dynamic causes the formal parameter `op` to be dynamic everywhere, the application of the method `eval` in `ListElm.fold` to be dynamic, and the classes `Binary`, `Add`, and `Mult` to be dynamic. The constraint that `init` is dynamic causes the formal parameter `init` to be dynamic everywhere. All other program parts are annotated static.

Specialization for the list value given by the substitution

$$\rho_2 = [\texttt{new ListElm}(6, \texttt{new ListElm}(4, \texttt{new Empty}()))/\texttt{ls}]$$

yields the residual program of Figure 8.16. The recursive call in the method `fold` has been unfolded, and all references to the classes `List`, `ListElm`, and `Empty` have disappeared. The `eval` methods of the classes `Binary`, `Add`, and `Mult` are specialized speculatively for each recursive iteration of `fold`. Each recursion of the method `fold` is unfolded into the calling context (a speculatively specialized version of an `eval` method) and gives rise to a new specialized `eval` method.

```
class Binary extends Object {
  Binary() { super(); }
  int eval( int x, int y ) {
    return this.eval( lift(x), y );
  }
}
class Add extends Binary {
  Add() { super(); }
  int eval( int x, int y ) {
    return lift(x)+y;
  }
}
class Mult extends Binary {
  Mult() { super(); }
  int eval( int x, int y ) {
    return lift(x)*y;
  }
}
```

```
class List extends Object {
 List() { super(); }
 int fold( Binary op, int init ) {
  return this.fold(op,init);
 }
}
class Empty extends List {
 Empty() { super(); }
 int fold( Binary op, int init ) {
  return init;
 }
}
class ListElm extends List {
 int i; List next;
 ListElm( int i, List next ) {
  this.i=i; this.next=next;
 }
 int fold( Binary op, int init ) {
  return
   op.eval(this.i,
           this.next.fold(op,init));
 }
}
```

```
 ls.fold(op,init)
```

Figure 8.15: Binding-time annotated version of the fold example with static list

```
aspect {
  introduction Binary {
    int eval_2(int y) { return this.eval_2(y); }
  }
  introduction Add {
    int eval_2(int y) { return 6+this.eval_1(y); }
    int eval_1(int y) { return 4+this.eval_0(y); }
    int eval_0(int y) { return y; }
  }
  introduction Mult {
    int eval_2(int y) { return 6*this.eval_1(y); }
    int eval_1(int y) { return 4*this.eval_0(y); }
    int eval_0(int y) { return y; }
  }
  main op.eval_2(init)
}
```

Figure 8.16: Specialization for list {6,4}

```
class Binary extends Object {
  Binary() { super(); }
  int eval( int x, int y ) {
    return this.eval( x, y );
  }
}
class Add extends Binary {
  Add() { super(); }
  int eval( int x, int y ) {
    return x+y;
  }
}
class Mult extends Binary {
  Mult() { super(); }
  int eval( int x, int y ) {
    return x*y;
  }
}
```

```
class List extends Object {
 List() { super(); }
 int fold( Binary op, int init ) {
  return this.fold(op,init);
 }
}
class Empty extends List {
 Empty() { super(); }
 int fold( Binary op, int init ) {
  return lift(init);
 }
}
class ListElm extends List {
 int i; List next;
 ListElm( int i, List next ) {
  this.i=i; this.next=next;
 }
 int fold( Binary op, int init ) {
  return
   op.eval(this.i,
           this.next.fold(op,init));
 }
}
```

```
   ls.fold(op,init)
```

Figure 8.17: Binding-time annotated version of the fold example with static operator and neutral value

Scenario 2: static operator and initial value

To specialize for static operator and initial value arguments, we provide the following initial constraint on the binding times of the free variables of the main expression:

$$\tau_3 = \{T_{\square,\square,\texttt{ls}} = D, T_{\square,\square,\texttt{op}} = S_N, T_{\square,\square,\texttt{init}} = S_L\}$$

We constrain the list argument to be dynamic, and the operator and initial value arguments to be static. The resulting binding-time annotation is illustrated in Figure 8.17. The constraint that the list is dynamic causes all invocations of the method `fold` to be annotated dynamic, and the classes `List`, `Empty`, and `ListElm` to be considered dynamic. All other program parts are considered static.

Specialization for the binary operator and initial value given by the substitution

$$\rho_3 = [\texttt{new Add()}/\texttt{op}, 0/\texttt{init}]$$

yields the residual program of Figure 8.18. The recursive calls to the method `fold` have been speculatively specialized, and specialized versions of the `fold` method have been generated. The calls to the `eval` method have been unfolded, and placed in the specialized `fold` methods.

8.8 Dealing with Object-Based Languages

The basic principles of partial evaluation for class-based languages developed in this chapter can be applied to object-based languages. The binding-time

```
aspect {
  introduction Client {
    int main_0( List ls ) { return ls.fold_add(); }
  }
  introduction List {
    int fold_add() { return this.fold_add(); }
  }
  introduction Empty {
    int fold_add() { return 0; }
  }
  introduction ListElm {
    int fold_add() { return this.i+this.next.fold_add(); }
  }
  main (new Client()).main_0( ls )
}
```

Figure 8.18: Specialization for operator `Add` and initial value `0`

analysis and specialization phases are similar to what we have defined for class-based languages, but the appearance of the specialized program is significantly different. We consider as future work the development of a complete partial evaluator for an object-based language; here, we briefly sketch how it can be accomplished.

This section discusses partial evaluation for object-based languages, gives an overview of a specialization phase for an object-based language, and last describes how binding-time analysis is done for an object-based language.

8.8.1 Considerations

To explain partial evaluation for object-based languages, we use a language similar to EFJ, named Object-Based EFJ, or OB-EFJ. The basic expressions are the same, except that there are no `new` expressions, and that there is an operation for creating a new attribute-less object, an operation for cloning an object, and a method assignment operation. There are no class declarations, but there are type declarations similar to class declarations that constrain where an object can be used, giving static typing similar to EFJ (we refer the reader to Abadi and Cardelli's work [1] for more details on object-based languages and type systems for object-based languages).

The considerations about partial evaluation for object-oriented languages that we presented in Chapter 5 hold for object-based languages, except that we only have described the representation of specialized programs for class-based languages. To recapitulate: partial evaluation specializes the interaction that takes place between objects by specializing the methods of these objects to their static arguments.

8.8.2 Specialization

In OB-EFJ, a virtual dispatch can be specialized like a virtual dispatch in EFJ: when the receiver object is static, the receiver method is unfolded into the caller; when the receiver object is dynamic, each potential receiver method is specialized speculatively, and a virtual dispatch to these methods is residualized. As was the case for class-based languages, it is only in a statically-typed language that a safe set of potential callees can be predicted. Creation of an empty object can be removed by specialization when the object is only used in static contexts. Object cloning can be removed by specialization when the object to be cloned is static and the resulting object only is used in static contexts. For simplicity, we may restrict methods to only appear on the right-hand-side of a method assignment. In this case, method assignment can be specialized like a field assignment to a constant value; if the self is static, the assignment is eliminated.

A specialized method must be present in an object at the program point where the generic method would have been called. For a class-based language, we simply add a specialized method to the class that it has been generated for, so that it is present when the specialized program is evaluated. In an object-based language, the specialized method can be introduced into the object using method assignment. However, the method must be assigned at some point during the evaluation of the specialized program. A specialized method could be introduced into the object at the call site where it is needed, but doing so could introduce a major overhead into the program; the call site could conceivably be in a loop or in a recursive function. Rather, the specialized method should be introduced into the object at the program point where the generic method was introduced into the object. We conjecture that in a statically-typed object-based language, a specialized version of a method can be assigned to any object where the generic method could be assigned.

With this specialization approach, the weaver functionality from class-based languages is embedded directly into the specialized program, and there is no obvious way of separating the specialized code from the generic code. As an alternative to this approach, an aspect language that permits introduction of methods into objects could have been used, but to the author's knowledge no such language exists.

8.8.3 Binding-time analysis

Binding-time analysis for OB-EFJ can be done similarly to binding-time analysis for EFJ, except that more elements from binding-time analysis for higher-order functional languages [113] may be needed. A control-flow analysis [146] or a type inference algorithm [2] can be used to determine the possible callees at a given call-site, as can a simpler analysis such as class-hierarchy analysis [50].

8.9 Summary

We have formally defined a small object-oriented class-based language, EFJ, and described its extension into a two-level language. We express specialization of EFJ programs in terms of evaluation of a two-level program, and give a binding-time analysis for automatically deriving a two-level program from a standard EFJ program. This minimal partial evaluator illustrates the basic principles of partial evaluation for class-based object-oriented languages. Partial evaluation for object-based languages is different, but can be done according to the same principles.

Our formalization of partial evaluation for object-oriented languages is essential for any future correctness proof, characterizes the essence of partial evaluation for the object-oriented paradigm, and facilitates reasoning about extensions to our basic partial evaluation approach.

Chapter 9

Partial Evaluation of Realistic Object-Oriented Programs

9.1 Introduction

The partial evaluation principles presented in the previous chapters describe the essence of partial evaluation for object-oriented languages. Nevertheless, these principles are insufficient to specialize realistic applications, and must be extended to enable partial evaluation to specialize realistic applications written in an object-oriented language.

There is a tension between simplicity and completeness in the design of a partial evaluator. The simplicity offered by a minimal set of analysis and transformation rules helps the user to predict the result of applying partial evaluation to a program, but limits the set of specialization opportunities that can be exploited using the partial evaluator. The partial evaluator can be extended to specialize a more complete set of programs, but at the cost of making the partial evaluator more complicated.

A partial evaluator can be designed with specific target programs in mind, and only include features needed to specialize these programs [41]. Our interest is to specialize object-oriented programs designed according to standard object-oriented techniques, and to eliminate overheads due to the use of these techniques. This goal determines a minimal set of features that we need from a partial evaluator; the features of the partial evaluator must match the programming conventions that exist in object-oriented languages. Concretely, a high degree of precision is needed from the binding-time analysis to match the potential for code-reuse, and common language abstractions must be supported by the analysis and transformation framework. Furthermore, to specialize realistic applications, a number of features known from partial evaluation for functional and imperative languages are needed.

This chapter describes the features that we have found essential when using partial evaluation to specialize realistic object-oriented programs. To a large extent, we have recast existing principles from partial evaluation for functional and imperative languages to work with object-oriented languages. For example, polyvariant binding-time analysis is well-known both from functional and imperative languages, and strategies for dealing with closures in functional languages can be used to extend the possibilities for specialization of object-oriented programs. The features described in this chapter were developed in conjunction with practical experiments with the partial evaluator JSpec presented in the next chapter. In this chapter we concentrate on class-based object-oriented languages; to investigate the features needed for partial evaluation of realistic programs written in an object-based language, a complete partial evaluator for such a language would have to be developed.

Chapter overview: This chapter first discusses analysis features; Section 9.2 presents use-sensitivity in the context of object-oriented languages, and Section 9.3 evaluates polyvariant analyses for object-oriented partial evaluation. Then, Section 9.4 proposes a means for exploiting information about how an object is initialized. Last, Section 9.5 discusses modular specialization for object-oriented languages, Section 9.6 discusses post-specialization optimizations for object-oriented languages, and Section 9.7 summarizes this chapter.

Additional language features and object-oriented partial evaluation

The concepts presented in this chapter and the examples used to illustrate them rely on some language features that are found in a complete language such as Java, but that are not found in our EFJ language. In this chapter, we add imperative features, class members, and abstract classes to EFJ. A class member is a field or a method that is defined independently of individual object instances, similar to a global variable or procedure in an imperative language. An abstract class contains undefined abstract methods, and cannot be instantiated; it serves to define an interface for its concrete (non-abstract) subclasses.

To specialize programs that use these language features, the partial evaluator described in the previous chapter has to be extended, as follows:

Imperative features: For the purpose of the examples given in this chapter, operations that perform side-effects with static values are removed by specialization; operations that perform side-effects with dynamic values are residualized; and loops are specialized by unrolling.

Class members: A use of a class field is removed by specialization when static, and residualized when dynamic. A call to a class method is specialized like a virtual method invocation with only one callee and a static self object. In a language with class fields and access modifiers, a virtual method with a static self object that accesses a dynamic class field declared private cannot be unfolded into a caller from a different class; it must be residualized as a class method in the same class. In general, to offer a finer degree of control over the size of methods in the specialized program, a virtual method specialized for a static self object can always be residualized as a class method; subsequent inlining optimization can be applied to optimize the specialized program where possible.

```
Power cube = new Power(x,y,z);
int i = cube.raise(2);
System.out.println(i);
System.out.println(cube);
```

(a) Printing an object and an integer obtained from the object.

```
Power cube = new Power(x,y,z);
int i = cube.raise(lift(2));
System.out.println(i);
System.out.println(cube);
```

(b) Dynamic annotation implies dynamic computation.

```
Power cube = new Power(x,y,z);
int i = cube.raise(2);
System.out.println(lift(i));
System.out.println(cube);
```

(c) Use-sensitive specialization.

```
Power cube = new Power(x,y,z);
System.out.println(8);
System.out.println(cube);
```

(d) Specialized program.

Figure 9.1: Use-sensitivity of binding times.

Abstract classes: Abstract methods are ignored during analysis, but must be taken into account during the specialization phase. A virtual method invocation that is specialized for a dynamic self object may include an abstract method as a potential callee. In this case, a dummy method that generates an error when called is introduced into the class of the abstract method, to allow the concrete specialized methods in the subclasses to be called. (An abstract method cannot be used, since it would force any subclass to implement the specialized method, even those not included in the set of possible callees.)

These language features are found in most class-based object-oriented languages.

9.2 Use-Sensitivity

A binding-time analysis assigns binding times based on information about the values in the execution context of the program. Nevertheless, the binding time of a given computation may depend not only on whether the computation uses dynamic data, but also on how the result of the computation is used elsewhere in the program. In an object-oriented program, the binding times of objects can exhibit such dependencies, which may result in overly conservative binding times.

This section presents the concept of use-sensitivity, which permits each use of an object to be given an individual binding time. First, use-sensitive binding times are introduced, and their importance for object-oriented languages is discussed. Then, it is described how to compute use-sensitive binding-time information, and how to specialize a program in a use-sensitive manner.

9.2.1 Use-sensitive binding times

The specialization action taken for the uses of a given value is directly linked to the binding time of the value: if the value is static, all its uses are removed by specialization; if the value is dynamic, all its uses are residualized. However, the context in which a computation appears may influence its binding time. A computation that appears in a dynamic context must be residualized in this context; if the computation produces a value that cannot be residualized, it is forced to be dynamic, even if the values that it depends upon are known. Residualization of values is done using the lift operator, which allows a base type value to be residualized, but not compound type values such as objects. As an example, Figure 9.1a shows a program that creates an object of class `Power` that is used to compute a simple value. Both the object and the value are printed to standard output. We assume that I/O operations are dynamic, so the argument to each call to `System.out.println` appears in a dynamic context. Thus, both the object and the value must be residualized, which forces the object to be annotated dynamic.

Since an object cannot be residualized, a known object that appears in a dynamic context must be annotated as dynamic to ensure consistent binding times. With a standard binding-time analysis, if an object is annotated dynamic at one program point, it must be annotated dynamic throughout the entire program. In a program where objects are used abundantly, this restriction on when objects can be considered static can be a serious limitation to successful specialization of a program. This problem is illustrated in Figure 9.1b. We use the binding-time inference rules of the previous chapter, and underlining indicates dynamic binding-time annotation. The use of the object `cube` in a dynamic context forces all computations involving the object `cube` to become dynamic, which causes the specializer to reproduce the original program unchanged.

We can allow the specialization actions for a given value to be sensitive to each use of the value. This feature is referred to as *use-sensitive* specialization [77, 79]. Use-sensitive specialization allows the computation of a value to both be performed during specialization and residualized in the specialized program. In effect, such computations are considered both static *and* dynamic. We extend our binding-time domain to include a new binding time named static-and-dynamic. Intuitively, a computation that depends on static (or static-and-dynamic) information is annotated static-and-dynamic when a value that it produces is used in a dynamic context.

Figure 9.1c shows binding-time annotations with use sensitivity; program parts with static-and-dynamic binding times are printed in italics. The expression that produces the `Power` object is annotated as static-and-dynamic, since this object is used both in a static context (the computation of the integer `i`) and in a dynamic context (the second call to `println`). Specialization according to these binding-time annotations produces the program of Figure 9.1d, where the power computation has been eliminated by the partial evaluator.

9.2.2 Use-sensitivity for object-oriented languages

Use-sensitivity increases the precision of the binding-time analysis. A program part that would otherwise have been annotated as dynamic can be annotated as static-and-dynamic, which enables specialization based on values produced by this program part. But is use-sensitivity essential to partial evaluation for object-oriented languages? The binding-time analysis presented in the previous chapter did not include use sensitivity, and could still be used to specialize programs.

Use sensitivity lets each use of an object be treated as a separate case, which has several implications beyond the example of Figure 9.1. First, each reference to a field of an object can be treated separately and thus be given separate binding times. This feature enables partially static objects to be effectively specialized, i.e., to handle objects where some fields have static values and some fields have dynamic values. The object itself is given a static-and-dynamic binding time, references to static fields are removed by specialization, and references to dynamic fields are residualized. The second implication of use-sensitivity is that a virtual dispatch can be specialized for a static self object when the self object is known to the partial evaluator, independently of whether the self object appears in a dynamic context. Thus, use-sensitivity can increase the number of virtual dispatches that can be simplified using partial evaluation. For object-oriented partial evaluation, use-sensitivity fulfills a functionality similar to the lift operator, but for object values; it allows specialization over objects that appear in a dynamic context.

As an example, consider a computation over complex numbers with an absolute value operation `abs`, as follows:

```
class Complex { float r,i; float abs() { return Math.sqrt(r*r+i*i); } }
... Complex x = ...
    ... graph.plot( x.r, x.i, x.abs() ) ...
```

The complex number `x` is plotted using an external routine invoked using the call "`graph.plot`." Suppose that the real part of the `Complex` instance `x` is dynamic and that the imaginary part is static. Specialization of this code fragment with an imaginary value of `2.0` yields the following specialized code:

```
aspect X { introduction Complex { float abs_s() {return Math.sqrt(r*r+4.0);} }
          ... Complex x = ...
              ... graph.plot( x.r, 2.0, x.abs_s() ) ... }
```

All references to the field `r` are residualized, and all references to the static field `i` are removed by specialization. The call `x.abs` has been specialized even though `x` appears in a dynamic context (the field reference `x.r`).

9.2.3 Use-sensitive binding-time analysis

Use-sensitive binding-time analysis has been proposed both as a data-flow analysis and as a constraint-based analysis. The data-flow version is used in the Tempo partial evaluator for C [77, 79], and the constraint-based version is used in a partial evaluator for a small functional language [8] (use-sensitivity is here referred to as "both static and dynamic expressions"). As presented, the data-flow approach provides a higher level of precision, but both approaches are equally applicable to object-oriented languages. The Tempo data-flow analysis is known to scale to large programs written in a realistic language, and is used in the JSpec partial evaluator for Java (see next chapter).

9.2.4 Use-sensitive specialization

Specialization of an object-oriented program annotated with use-sensitive binding times is mostly straightforward. Specialization of an expression annotated static-and-dynamic will both produce a value to be used for further specialization and a specialized expression to be residualized in the specialized program. Most language constructs specialize straightforwardly using static-and-dynamic binding times. For example, a field reference annotated as static-and-dynamic produces a value during specialization, but also appears in the specialized program. The only complication is specialization of virtual method invocations.

A method invocation over a static self object is unfolded since the reference to the self object will disappear, whereas a method invocation over a dynamic self object is residualized as a virtual method invocation to a specialized version of each possible callee. For a method invocation annotated as static-and-dynamic, the callee method is known during specialization, but the reference to the self object must be residualized along with the non-static parts of the callee method. Specialization generates a single specialized method of a unique name for the concrete class of the object. In a dynamically-typed language, this method can simply be called. However, in a statically-typed language, an invocation of this specialized method might be illegal: the concrete class for which the method is generated may be a subtype of the qualifying type, in which case the specialized method is not visible at the call site.

There are three immediate solutions to the illegal method call problem. First, a type cast could simply be inserted to cast from the qualifying type to the concrete type; there is only a single possible callee, and the virtual dispatch is easily removed by an optimizing compiler. (In a language that supports direct method calls, a direct call to the specialized method could simply have been used, but a type cast is still needed.) Second, a dummy method can be introduced into the class of the qualifying type, similar to calls involving abstract methods. However, the information that there is only a single possible callee is lost, and the virtual dispatch is no longer easy to remove. Third, the virtual dispatch can be unfolded into the caller. However, unfolding may cause residual references to dynamic fields of the receiver object to appear in the sender. These fields must be accessed using a type cast, and to access them outside of the receiver object may be illegal in a language with access modifiers. In any case, we prefer inlining in a post-processing phase over unconditional unfolding, as explained in Section 9.6. Of these three options, we prefer the simple strategy of using a type cast.

9.3 Polyvariant Analysis

When a single program part is used in multiple places throughout a program, it can be advantageous to analyze the program part multiple times, once for each specific usage context. Polyvariant analyses can derive multiple analysis results for the same program part, and are essential for determining precise information about program parts that are heavily reused. In the context of object-oriented languages, code reuse is often cited as one of the primary software engineering advantages of this language paradigm. In an object-oriented language, individual classes can be created and then joined together to form a complete program. A class can be developed as an encapsulated program unit that implements a general functionality. This approach permits the class to be directly reused for different purposes. This style of code reuse must be taken into account in the design of a partial evaluator: two objects of the same class may require different treatment by the partial evaluator.

In this section, we present various polyvariant analyses that are essential for a precise binding-time analysis of an object-oriented program. The concept of polyvariance applies to different entities such as types, methods and aliases; we will explain each of these in turn. However, polyvariance may result in overly detailed information which can lead to over-specialization; when applied to a large program, polyvariance must be controlled.

9.3.1 Type-polyvariant binding times

One of the advantages of object-oriented programming languages is the ease with which classes can be used to introduce new data types into a program. Data that serve a single purpose can naturally be treated uniformly by the partial evaluator across the entire program, whereas other kinds of data must be treated individually for each object instance. For example, when objects are used to represent complex numbers or high-precision bignums, these objects should be treated individually, just like standard integers are treated individually. As for container classes such as stacks or lists, the binding time of a container should depend on the binding times of the elements that it contains, not be globally uniform for all data structures of the same class. Indeed, in partial evaluators for functional languages lists usually have individual binding times [22, 40], as do arrays in imperative languages [6, 41].

As an example, consider a container class `Box`. An object of class `Box` has some other object as its content:

```
class Box { Object c; Box( Object x ) { c=x; } }
```

Objects of different binding times can be stored in two different objects of class `Box`, but type-monovariance would force their binding times to become the least-upper bound of the individual binding times:

```
Box b1 = new Box( static_object ), b2 = new Box( dynamic_object );
 ... b1.c ...
```

Here, type monovariance would force the object `static_object` to become dynamic when stored in an object of class `Box`. As a result, the object obtained through `b1.c` would also be dynamic.

For simplicity, the binding-time analysis presented in the previous chapter enforced the same binding times for every instance of a given class. To generalize the binding-time analysis to be type-polyvariant, an individual binding time should be associated with each object value computed at a given program point. Type-polyvariant binding times are well-known from functional languages, and are for example supported by the partial evaluator Schism [39]. In practice, it can be useful to allow the user to choose between type-polyvariant and type-monovariant binding times on a class-by-class basis. Type-monovariant binding times offer a simple means of controlling the binding-time analysis when all instances of a given class should contain values with the same binding time.

9.3.2 Method-polyvariant binding times

Type-polyvariant binding times permit the data stored within each instance of an object to be given individual binding times. This data is manipulated and made visible outside the object by operations performed on the object, such as field access and method invocation. For the type-polyvariant binding times to be useful, the binding times of these operations must depend on the binding times of the individual object instance. Field access is usually a basic operation in the language. Hence, it can directly be given a binding time that depends on the binding time of the object that is accessed at a given program point. However, method invocation involves binding the self object and the formal parameters of the method to a given object; to make the binding time of each method invocation depend on the binding times of the object that is accessed at each program point, the binding time of the self object and the formal parameters must vary with each method invocation.

As an example, consider the container `Box` extended to have an operation for obtaining the content:

```
class Box { Object c; Box( Object x ) { c=x; } Object get() { return this.c; } }
Box b1 = new Box( static_object ), b2 = new Box( dynamic_object );
 ... b1.get() ... b2.get() ...
```

To have individual binding times for the objects stored in `Box` objects, it is no longer enough that each instance of `Box` has individual binding times; each call to the `get` method must also be treated individually.

A method-polyvariant binding-time analysis permits individual binding times to be assigned to each method invocation, by creating multiple variants of a method; each variant is analyzed individually with its binding-time context found at a method invocation site. Method-polyvariant binding-time analysis for an object-oriented language is mostly equivalent to a standard polyvariant binding-time analysis for a functional or imperative language, which is well documented in the partial evaluation literature [39, 78].

9.3.3 Polyvariant alias information

A partial evaluator for a language with imperative features relies on an alias analysis to determine what locations may be read and written at each program point [6]. The binding times of the program are computed based on the binding times of the locations manipulated by the program. Thus, the precision of the binding-time analysis is influenced by the precision with which the alias analysis traces what locations are read and written at each program point. Concretely, a monovariant alias analysis treats all objects passed to a given method as being aliased; this situation defeats the purpose of method polyvariance. To have unrestricted type-polyvariant and method-polyvariant binding times in a language with imperative features, a type-polyvariant and method-polyvariant alias analysis is needed.

As an example, consider the annotations that a monovariant alias analysis would derive for the previous example with the Box container:

```
class Box { Object c; Box( Object x ) { c=x; }
            Object get() { return this<Box1,Box2>.c; } }
Box b1 = new<Box1> Box( static_object<SO> ),
    b2 = new<Box2> Box( dynamic_object<DO> );
 ... b1<Box1>.get()<SO,DO> ... b2<Box2>.get()<SO,DO> ...
```

Alias annotations are indicated using the notation <L>, where L is the name of an abstract location. Outside the class Box, the locations Box1 and Box2 are separate and their fields can have separate aliases (the locations SO and DO). However, inside the method get of class Box, the alias annotations are merged since there is a unique alias variant per method. Thus, the results obtained from each call to get are merged, and both SO and DO are aliases.

Method-polyvariant alias analysis is often considered prohibitively expensive and, in the context of optimizing compilers, not worth the extra complexity [75, 140]. However, partial evaluation is designed to target a specific program slice; as a result the program can be analyzed in detail and subsequently aggressively optimized. Thus, the use of a polyvariant alias analysis is feasible in a partial evaluator. The practical experiments conducted with the JSpec partial evaluator (described in Chapters 10 and 11) using a monovariant alias analysis reveals that although specialization is possible, alias monovariance makes specialization of many interesting programs impractical.

Binding-time analysis for an object-oriented language relies on an analysis to compute the possible callees at each call site. This information can be computed using an alias analysis; the alias analysis infers the set of objects that may appear at the call site, which is an estimation of the set of possible callees. If there is only a single possible callee, the virtual call can be eliminated. This use of alias information can be seen as a form of specialization. Indeed, this behavior is exhibited by the alias analysis phase of the JSpec partial evaluator. When the alias analysis is polyvariant, each invocation of a method is specialized to alias information about its arguments by eliminating virtual dispatches when possible. This way, multiple alias variants of a method are generated during partial evaluation; each variant may be unique in terms of what virtual dispatches have been eliminated, and may thus be generated as a separate specialized method in the residual program. The alias analysis is based on initial alias information supplied by the user together with the initial binding-time information; the program is thus specialized, not only to the binding-time information, but also to the alias information. To control the amount of specialization due to alias information, the precision of the alias analysis must be controlled.

9.3.4 Controlling polyvariance

Method variants are generated, not only by a polyvariant alias analysis, but also by the binding-time analysis. In addition, when using type-polyvariance, method variants may be generated for each object allocation point in the program. In general, the use of polyvariance can cause an explosion in the number of method variants. An excessively high number of method variants can be a major overhead during analysis and can lead to code explosion in the specialized program. For this reason, the number of generated variants must be carefully controlled. For an object-oriented language, a solution consists in allowing the user to declare which classes to treat with a high precision (polyvariant analyses), and then to leave all other classes of the program to be treated with low precision (monovariant analyses).

9.4 Instantiation-Time Specialization

The standard partial evaluation strategy for specializing procedures in an imperative language is to specialize a procedure when it is called; we have chosen the same strategy for methods in object-oriented languages. However, information that was known about an object when it was instantiated may have been lost by the partial evaluator when methods are invoked on the object. In this case, it can be advantageous to speculatively specialize these methods for the known information on how the object was initialized.

This section presents the concept of *instantiation-time specialization*, which allows methods to be specialized to object information available at object initialization time rather than information available at method invocation time. First, the basic concept is introduced and described in detail. Then we describe the changes needed in a partial evaluator to implement instantiation-time specialization. Afterward, safety issues are discussed, and finally some perspectives on instantiation-time specialization are given.

9.4.1 Instantiation-time specialization: what

In an object-oriented language, a method can be specialized to its known arguments at each invocation site; specialization adapts the method to the value of the self object and the value of the parameters. However, an object that is initialized using known information may have become dynamic when it is the subject of a method invocation. With a flow-sensitive binding-time analysis, this situation can for example occur when static and dynamic objects are mixed together in a container, or when a choice between several static objects is done

```
class Binary extends Object {
  Binary() { super(); }
  int eval( int x, int y ) {
    return this.eval( x, y );
  }
}
class Add extends Binary {
  Add() { super(); }
  int eval( int x, int y ) {
    return x+y;
  }
}
class Mult extends Binary {
  Mult() { super(); }
  int eval( int x, int y ) {
    return x*y;
  }
}
```

```
class Power extends Object {
  int exp; Binary op; int neutral;
  Power( int exp, Binary op, int neutral ) {
    super();
    this.exp = exp;
    this.op = op;
    this.neutral = neutral;
  }
  int raise( int base ) {
    return loop( base, exp );
  }
  int loop( int base, int e ) {
    return e==0
     ? this.neutral
     : this.op.eval( base,
                     this.loop( base, e-1 ) );
  }
}
```

```
Power[] p = new Power[10];
for( int i=0; i<10; i++ ) p[i] = new Power(i,new Mult(),1);
...
... p[dyn_1].raise(dyn_2) ...;
```

Figure 9.2: Statically constructed object used under dynamic control.

based on dynamic information. It is often the case that more is known about an object when it is initialized than when its methods are invoked.

As an example, consider the use of the `Power` class in Figure 9.2. The binary operators and the `Power` class are defined as in the previous chapters. The `Power` class is used in a program fragment where 10 different instances of `Power` are created in a loop, and then used at a later program point. When the `Power` objects are created, the information needed to specialize the method `raise` is static. However, this information has become dynamic when the method `raise` is actually invoked, since the reference to the power object has become dynamic.

When the fields of an object are not modified between the point where the object is instantiated and the point where some method is invoked on the object, this method can be specialized to the values of these fields. As long as the fields remain unchanged, the specialized method can safely be used in place of the generic method. Specialization of a method to values supplied at object instantiation time is advantageous if the method could not otherwise have been significantly specialized. Conversely, this approach may be disadvantageous in the case where specialization could have been done at method invocation time, since the values of the formal parameters are not available at instantiation time.

Returning to the `Power` example of Figure 9.2, we want a specialized version of the `raise` method for each object instance. Each specialized method should be generated based on the values of the static fields of the self object, and should be automatically used at the invocation site of the method `raise`. Conceptually, methods should be speculatively specialized for the context in which the object is instantiated, and then used in place of the methods that would otherwise have been called at a specific application site. We refer to this specialization strategy as instantiation-time specialization, or ITS for short.

9.4.2 Instantiation-time specialization: how

We refer to the methods that are specialized using ITS as the ITS target methods, and refer to their class as the ITS target class. To perform ITS, the ITS target methods must be analyzed with the newly instantiated object as self object and dynamic values for all other parameters. The specialized methods that are generated should be used at every call site where the corresponding methods are invoked. In a language without side-effects, the ITS target methods can be specialized for the state of the object just after instantiation. However, in a language with side-effects, the side-effects of the ITS target methods would incorrectly be considered to have taken place at object instantiation-time, before the methods actually were called. Thus, the partial evaluator must save the current state of the program before the methods are specialized, and restore the program state afterwards.

Once the ITS specialized methods have been generated, they must be integrated back into the program so that they will be called when the ITS target methods would otherwise have been called. This behavior can be implemented using inheritance; the specialized methods are placed in a newly generated subclass of the ITS target class, and this newly generated subclass is instantiated in place of the ITS target class. At the method invocation site, an ordinary virtual dispatch selects which ITS-specialized variant of a method to use. To avoid breaking encapsulation in the case where the specialized methods need to access dynamic state stored in the object, the methods in the subclass can be wrappers that call specialized methods introduced into the ITS target class.

ITS works best in a language with constructors (or a similar mechanism), since it in such a language is explicit when an object has been initialized. In a language without constructors, the user must indicate at what program point an object that is to be specialized using ITS has been initialized.

As an example of instantiation-time specialization, Figure 9.3 illustrates how the generic program shown in Figure 9.2 can be specialized using ITS. The specialized `raise_`*x* methods are introduced into the class `Power`, and ten subclasses of `Power` are introduced into the program: one for each specialized method (we assume that classes can be declared locally to an aspect, similar to Java inner classes). Each new subclass contains a constructor that directly calls the superclass constructor, and a method `raise` that calls the corresponding specialized `raise_`*x* method in the superclass. The object instantiation expressions have been changed to now instantiate the appropriate subclasses.

9.4.3 ITS implementation

ITS can be implemented in a method-polyvariant partial evaluator for a class-based object-oriented language with a minimal amount of changes, as follows. The partial evaluator places calls to the ITS target methods at the end of the constructor (or just after the object has been initialized). The ITS target methods are then analyzed in the correct context, and can after specialization be extracted from the specialized program by the partial evaluator.

In a language with side-effects, the program state must be saved before

```
aspect ManySpecPower {
 introduction Power {
  int raise_0( int base ) { return 1; }
  int raise_1( int base ) { return base; }
  int raise_2( int base ) { return base*base; }
  ...
  int raise_9( int base ) {
   return base*...*base;
  }
 }
 class Power_0 extends Power {
  Power( int exp, Binary op, int neutral ) {
   super( exp, op, neutral );
  }
  int raise( int base ) {
   super.raise_0(base);
  }
 }
 class Power_1 extends Power {
  Power( int exp, Binary op, int neutral ) {
   super( exp, op, neutral );
  }
  int raise( int base ) {
   super.raise_1(base);
  }
 }
 ...
 class Power_9 extends Power {
  Power( int exp, Binary op, int neutral ) {
   super( exp, op, neutral );
  }
  int raise( int base ) {
   super.raise_9(base);
  }
 }
```

```
 introduction ... {
  ...(...) {
   Power[] p = new Power[10];
   p[0] = new Power_0(0,new Mult(),1);
   p[1] = new Power_1(1,new Mult(),1);
   p[2] = new Power_2(2,new Mult(),1);
   p[3] = new Power_3(3,new Mult(),1);
   p[4] = new Power_4(4,new Mult(),1);
   p[5] = new Power_5(5,new Mult(),1);
   p[6] = new Power_6(6,new Mult(),1);
   p[7] = new Power_7(7,new Mult(),1);
   p[8] = new Power_8(8,new Mult(),1);
   p[9] = new Power_9(9,new Mult(),1);
   ...
   ... p[dyn_1].raise(dyn_2) ...;
  }
 }
}
```

Figure 9.3: Instantiation-time specialized version of Figure 9.2.

the ITS target methods are specialized, and restored afterwards. Saving and restoring program state is a standard operation in a partial evaluator that supports speculative specialization of dynamic conditionals. Specifically, the ITS target can be specialized without incorrect side-effects by placing their invocations inside a dynamically controlled conditional. This approach causes any locations side-effected by the methods to become dynamic after object instantiation. It is a safe approximation to make state dynamic, so we consider this drawback acceptable. Indeed, the partial evaluator JSpec presented in the next chapter implements ITS in this way.

9.4.4 Instantiation-time specialization safety

Instantiation-time specialization is straightforward in a language with immutable object state, since the state of an object is fixed when the object has been created. In a language with mutable object state, ITS is safe when the fields of an object are not modified after initialization of the object. Encapsulation and access modifiers facilitate ensuring field immutability as an invariant, since private fields only can be modified by local methods. As an alternative to immutable fields, fields can be guarded using the specialization class approach, to ensure that the ITS-specialized methods are used only when the invariants for which they were specialized hold. In both cases, we assume that the user explicitly indicates the classes and methods for which to perform ITS. As an alternative to these manually controlled approaches, a backwards analysis could determine which method invocations are done on dynamic objects free of side-effects on their state, and then only apply ITS to these objects. We consider the development of such an analysis as future work.

9.4.5 Perspectives on instantiation-time specialization

Instantiation-time specialization is directly inspired by the "frozen closure" feature of the Schism partial evaluator [40]. In Schism, closures that pass under dynamic control without being applied are specialized in-place for the values of their static free variables. Indeed, closures in functional languages are akin to objects; the free variables of a closure correspond to the construction parameters of an object; and the arguments given to a closure upon invocation correspond to the arguments given to a method upon invocation.

Specialization of a self-interpreter is a well-known benchmark for evaluating the effectiveness of a partial evaluator [84, 103, 114]. Ideally, a partial evaluator should completely eliminate a layer of interpretation; specialization of a self-interpreter to an input program should ideally reproduce the input program in the specialized program (modulo alpha renaming). Frozen closures play a central role in the specialization of interpreters for functional languages. As an example, consider an interpreter for a higher-order functional programs, where higher-order functions at the interpreted language level are implemented using higher-order functions, e.g., in Scheme:

```
; interpret-lambda: (environment,[parameter],exp) -> ([value] -> value)
(define interpret-lambda
  (lambda (env formals body)
    (lambda (arguments) (interpret (combine formals arguments env) body))))
```

This function interprets a lambda form in a given environment, by returning a Scheme function that represents the interpreter closure. When applied to a list of arguments, this Scheme function interprets the closure with the contents of the list as arguments; the complete interpreter is shown in Figure A.2 of the appendix. When a closure is created by interpreter using the function `interpret-lambda`, the function body is a known parameter (it is given by the program text, which is static). However, when the same closure is later applied to its arguments, the function body has become a value at the interpreter level:

```
; interpret-apply: (environment,exp,[exp]) -> value
(define interpret-apply
  (lambda (env e0 args)
    ((interpret env e0) (map (lambda (a) (interpret env a)) args))))
```

The closure is obtained by interpretation of the input program, and is therefore dynamic (like all values computed by the interpreted program). To specialize the interpreter for the function body, the closure must be specialized when it is created, and then used later in its specialized form.

The complications in specializing a self-interpreter for an object-oriented language are similar to those for self-interpreters of functional languages. Consider an interpreted language that defines classes that are instantiated into objects; these objects hold data and can receive messages (i.e., have methods that can be invoked). All interpreter-level classes can be represented as instances of the same implementation-level class; in particular, the methods of the interpreted program are all manipulated through instances of this implementation-level class. During interpretation, an implementation-level class is instantiated for each class definition in the input program. As was the case for functional languages, this program point is where the body of each method is known to the specializer. When a method is later invoked, the self object and its class are unknown, and no specialization can be done. Instantiation-time specialization allows concrete subclasses to be generated for each interpreter-level class; each such subclass contains a version of the interpreter specialized for the methods stored in the corresponding class. In the specialized program, a virtual dispatch in the implementation is used to select between these subclasses and thereby to determine the receiver class of a virtual dispatch at the interpreted level.

We conclude that partial evaluators for functional languages and object-oriented languages can specialize self-interpreters. Furthermore, we conjecture that closures or objects that link code and data are essential for specialization of a self-interpreter for a language with higher-order procedures. Thus, for the lack of closures or objects, self-interpreters for imperative languages with function pointers cannot be specialized.

9.5 Modular Specialization

The capability to specialize only a slice of a program rather than the entire program is essential when applying partial evaluation to specialize realistic applications [41]; this partial evaluator feature is referred to as modular specialization. The purpose of program slicing in conjunction with partial evaluation is three-fold: it allows the user to precisely specify invariants for which to specialize the program slice; it delimits the scope of specialization to prevent over-specialization; and it allows for a more efficient analysis since the whole program need not be analyzed.

This section discusses modular specialization for object-oriented languages. In a class-based object-oriented language, program slicing can be done at many different granularities; the finer the granularity, the more complications associated with implementing program slicing in the partial evaluator. Furthermore, for modular specialization to be easy to use, there should be a suitable default behavior for methods not included in the program slice.

9.5.1 Program slicing granularity

In an object-oriented language, program slicing can be done in terms of modules, hierarchies of classes, individual classes, or individual methods. For the purpose of partial evaluation, module-level slicing is often too coarse. A module can for example encompass all the features of a complete library; partial evaluation would usually be applied to specialize only a limited set of features in a library. A slice that encompasses a class and optionally all of its subclasses forms a logical unit that can be suitable in size for specialization. However, it is likely that not all the methods of a collection of classes should be specialized. In this case, it must be possible to specify whether each method is included; such a slice cross-cuts the class structure of the program, which complicates the slice specification.

When slicing is done at the class hierarchy level, either all receivers of a virtual dispatch are included in the program slice or none of them are; it is only in the former case that specialized methods can be generated. When slicing is done below the class hierarchy level, i.e., in terms of individual classes or methods, specialization of virtual calls can be impeded. Speculative specialization of a virtual call causes specialized methods of identical names to be generated and introduced into all potential receiver classes. However, if one of these receiver classes is not included in the program slice, then no specialized method can be generated. Hence, the virtual call cannot be specialized. The simplest solution is to residualize such a virtual call in its original unspecialized form, which is the solution taken in the JSpec partial evaluator.

9.5.2 Behavior of external methods

When specialization is performed on a program slice, symbolic information must be provided about the behavior of the surrounding program parts. A partial evaluator can define a default behavior for such enclosing program parts; an appropriate default behavior reduces the amount of information that must be supplied by the user, which simplifies the task of applying partial evaluation to a program.

As an example, consider the Tempo partial evaluator for C [41]. Tempo allows an external function (a function not included in the program slice that is specialized) to be symbolically described using standard C syntax. If no description is given, an external function is assumed to rely only on the immediate values of its parameters (i.e., the address of a pointer, not what it points to) and make no side-effects; any pointer return value is assumed to be a null-pointer. The default assumptions about side-effects and the return value are pragmatic: they are unsafe, but the only safe assumption in a language like C is that an external function can dynamically side-effect any location in the program; such an assumption would make the analysis results unusable.

For an object-oriented language, a useful default would be to assume that an external method reads the fields of the self object and returns a dynamic object matching the return type. To return an object rather than a null reference is essential if an alias analysis is used to determine the callees of a virtual dispatch over the returned object (as in Section 9.3). There are no potential callees for a virtual dispatch over a null reference, but an external method could return any object; invoking methods on this object could have dynamic side-effects. In a language with access modifiers, a more refined default behavior could perhaps be defined. However, the more complex the default behavior, the harder for the user to determine whether it is suitable for a given specialization scenario.

9.6 Post-Specialization Optimizations

Partial evaluation aggressively optimizes a program using specific transformations. Yet, a residual program often contains opportunities for optimization, in part due to the nature of the transformations performed by the partial evaluator, in part because partial evaluation does not optimize dynamic computations. For this reason, many partial evaluators post-process the residual program with an optimization pass designed to eliminate overheads that typically occur in residual specialized programs.

This section discusses post-specialization optimization in the context of object-oriented languages. First, we motivate the need for optimizations. Then, we describe optimizations that are useful in the context of object-oriented languages.

9.6.1 Post-specialization optimization

A specialized program produced by a partial evaluator has been aggressively optimized according to static input data. The optimizations that are applied typically include global constant propagation of all data types and generalized constant folding. However, many opportunities for optimization may, and usually do, remain. These optimization opportunities may be due to the way residual code is generated by the partial evaluator, limitations of the partial evaluator, or dynamic computations. Ideally, optimizations would be carried out during the compilation of the residual program to result in a completely optimized binary executable. However, it is often the case that standard compilers fail to properly exploit the opportunities for optimization that remain in residual programs. Compiler optimizations are expected to normally improve and never degrade program performance, which forces a compiler to be conservative. In the context of partial evaluation, additional information is available for optimization; for example, a program slice that has been specialized is usually critical for performance. Also, compiler optimizations are targeted towards human-written code, not code produced by the transformations of a specific partial evaluator. For these reasons, the performance of specialized programs can be greatly enhanced by incorporating a post-specialization optimization phase to aggressively optimize the residual code using standard compiler optimizations. For example, Tempo integrates a set of post-specialization optimizations for obtaining efficient specialized programs, including arithmetic simplifications, dead-code elimination, copy-propagation, and aggressive function inlining.

9.6.2 Optimizations for object-oriented languages

The Java experiments reported in Chapter 11 reveal that inlining is handled well by existing compilers for Java, but that there are numerous opportunities for optimization by object inlining and common subexpression removal. We would prefer that these optimizations be handled by a compiler, but the Java compilers used for experiments do not satisfactorily remove overheads that can be eliminated by these optimizations.

Partial evaluation followed by aggressive method inlining optimization often results in few methods that access many objects; control flow is simplified but data is still accessed via indirections. For example, object access methods, that before inlining were called as virtual methods, may have been inlined into the callee, but the data that they access are still accessed through object references. Concretely, consider a use of the class `Complex` implemented using a bridge design pattern (shown in Figure 7.6 of Chapter 7):

```
Complex c = ...;
double x = c.r()+c.i();
```

After specialization to the `RectImp` representation and subsequent inlining optimization, this program fragment becomes:

```
Complex c = ...;
double x = ((RectImp)c.imp).r + ((RectImp)c.imp).i;
```

Common subexpression elimination simplifies this program fragment into:

```
Complex c = ...;
RectImp tmp = ((RectImp)c.imp); double x = tmp.r + tmp.i;
```

Although simple in this case, the optimization is more difficult when numerous dereferencing operations are common subexpressions across different parts of a method. An alias analysis may be needed to determine that the common subexpressions are invariant, and few compilers perform detailed alias analyses. In addition, in a language like Java, side-effects performed by other threads of execution must be taken into account.

Common subexpression elimination reduces the number of indirect memory references, but the program could be optimized even further using object inlining. With object inlining, an object that is used locally to a method M can be "inlined" into M, by inlining all methods of the object into M and making all fields of the object be local variables in M [26]. The complex number example could be simplified into:

```
...
double x = RectImp_tmp_r + RectImp_tmp_i;
```

Where the two variables `RectImp_tmp_r` and `RectImp_tmp_i` represent the fields `r` and `i` of the object stored in the field `imp` the the complex number `c` (these variables are initialized in the preceding code). Elimination of virtual dispatches by specialization and subsequent inlining is likely to increase the number of objects that can be inlined, since it increases the chance for an object to be local to a method. To determine the objects that are local to a method, an escape analysis can be used [18, 37, 165].

The common subexpression elimination and object inlining optimizations serve a dual purpose: the number of dereferencing operations is reduced, and the number of class casts is reduced. Class casts are residualized for invocation of methods with a static-and-dynamic self object and are needed to cast the self object when a method has been inlined. In a language like Java where casts are checked at run time, these class casts can be a major overhead.

We believe that these optimizations also would be useful in post-specialization optimization for other object-oriented languages, but only experiments with partial evaluation of programs written in other languages can reveal if this is really the case. Nonetheless, it is likely that for many languages other than Java, compiler inlining is not sufficient. For example, in C++ methods are only inlined when they are explicitly tagged with an inline modifier. To ensure sufficient inlining optimization, a post-specialization optimization phase could insert inline modifiers on methods that were deemed worthwhile to be inlined. In general, compilers for a given language may not perform aggressive inlining at all, in which case a post-specialization optimization phase must perform aggressive inlining optimization.

9.7 Summary

In a binding-time analysis, use-sensitivity, method-polyvariance and type-polyvariance are necessary for providing precise analysis results about the usage of each individual object. Furthermore, in an imperative language, a polyvariant alias analysis is needed to properly trace individual objects. However, when used to determine program control flow, the alias analysis itself becomes a form of program specialization. In general, polyvariant analyses may derive overly detailed information, and must be carefully controlled to avoid over-specialization. In terms of specialization, the methods of an object can be specialized not only when they are called, but also when the object is instantiated, akin to how closures can be specialized in a functional language.

When targeted to realistic object-oriented applications, a partial evaluator must support modular specialization. The more fine-grained the program slicing supported, the tighter the user control over what is specialized; program slicing at a fine level does however raise complications when specializing object-oriented programs. As for the specialized object-oriented programs generated by the partial evaluator, aggressive method inlining, object inlining and common subexpression elimination are useful post-specialization optimizations.

The partial evaluation features described in this chapter are essential for specialization of realistic object-oriented programs. However, they are non-trivial to implement, and can be complicated to control. As always, when designing a partial evaluator for an object-oriented language, each feature should be carefully evaluated in the light of the target programs.

Chapter 10

Partial Evaluation for Java

10.1 Introduction

The development of partial evaluation has been a continuous interaction between principles and practice. The first formal partial evaluation principles were used in the development a partial evaluator [86], and the further development of partial evaluators has revealed the need for more advanced partial evaluation principles, with an ensuing implementation to test these principles in practice [85].

The partial evaluation principles and techniques for object-oriented languages that are presented in this document were developed in unison with a complete partial evaluator for Java, named JSpec. It is the JSpec implementation that has permitted us to identify the set of features needed to specialize object-oriented programs.

JSpec is an off-line partial evaluator for the Java language which integrates a wide range of state-of-the-art analysis and specialization features, and offers several input and output language options:

- JSpec treats the entire Java language excluding exception handlers and `finally` regions, and takes as input either Java source code, Java bytecode, or C native functions.
- JSpec is integrated with a compiler for an extended form of specialization classes, which allows the binding times of the program to be specified separately from the program using a declarative language.
- The JSpec binding-time analysis is method-polyvariant, type-polyvariant, use-sensitive, and flow-sensitive; object values and base type values are treated similarly throughout the analysis framework, which ensures a consistent behavior that does not impose arbitrary limitations on the specialization process.
- Specialized programs are generated either as Java source code, encapsulated in an AspectJ [168] aspect, C source code for execution in the Harissa [117] environment, or binary code generated at run time for direct execution in the Harissa environment.

- JSpec is applied to a user-selected program slice, which allows expensive analyses and aggressive transformations to be directed towards the critical parts of the program.

In essence, JSpec incorporates the partial evaluation techniques thus far presented, extended to deal with Java.

JSpec has been designed with an emphasis on re-use of existing technology. In particular, JSpec uses the Tempo specializer for C programs as its partial evaluation engine, which permits the user to take advantage of the advanced features found in a mature partial evaluator. C is used as intermediate language in JSpec; its low-level nature makes it possible to represent the semantics of Java programs.

This chapter describes all aspects of the design and implementation of the JSpec partial evaluator. We describe how partial evaluation can be performed for Java (based on the techniques of the preceding chapters), and how we have implemented JSpec by re-using existing Java compiler technology and C partial evaluation technology. Based on our description of the JSpec design, we evaluate the advantages and disadvantages associated with the design. In this chapter we do not address performance issues; performance is the subject of the next chapter.

Chapter overview: First, Section 10.2 presents an overview of the Java language. Then, Section 10.3 describes how the JSpec partial evaluator treats Java programs. Section 10.4 presents an overview of the JSpec architecture and provides a detailed technical description of how a Java program is specialized using JSpec. Section 10.5 evaluates the software engineering experience gained in the design and implementation of JSpec. Last, Section 10.6 presents a summary.

Note: An initial prototype of JSpec that only produced specialized programs in C was presented earlier by Schultz et al. [143]. In comparison, this chapter describes the complete Java-to-Java specialization process, presents an in-depth view of partial evaluation for Java and the structure of JSpec, and evaluates the design of JSpec.

Availability

The JSpec partial evaluator is publicly available from the JSpec homepage `http://www.irisa.fr/compose/jspec`. The distribution includes most of the programs used for experiments in the next chapter. At the time of writing, the JSpec source code is available upon request.

10.2 Overview of the Java Language

The basic features of Java have already been introduced in Chapters 2 and 8, in the form of Extended Featherweight Java. Java is a statically-typed, class-based, imperative, object-oriented language with single inheritance between classes. An object is instantiated from a class using a constructor that performs initialization operations. Objects are dynamically allocated, contain fields and can receive method calls according to their class. Class fields and class methods (referred to in Java as static fields and static methods, respectively) allow data storage and code to be defined independently of objects. Base type values include boolean and numerical types with standard operators; a base type value is not considered to be an object, and cannot be manipulated as an object reference. However, there is direct support for arrays both of object references and of base-type values. The control constructs include conditionals and loops, as well as labeled `break` and `continue` statements.

In addition to standard concrete classes, Java has abstract classes and interfaces. An abstract class contains a mix of concrete methods and abstract methods, whereas an interface only contains abstract methods. A class can extend only one class, but can implement multiple interfaces, which allows a class to adhere to a set of independent abstract interfaces. This approach avoids many of the complications commonly associated with multiple inheritance [29], since methods inherited from an interface are always abstract.

Apart from standard imperative features and object-oriented features, Java includes statements for generating and handling exceptions, as well as a statement for ensuring that a block of code is executed before control leaves a given region (`finally` blocks). In addition, reflection, dynamic loading and multi-threading are integrated into the language, in part through the standard library.

10.3 JSpec: A Partial Evaluator for Java

JSpec is a complete partial evaluator for Java, that operates according to the principles described earlier. It has been designed with specialization of realistic Java applications in mind, and has been used to perform the specialization experiments reported throughout this document. We have in the preceding chapter already presented most of the techniques necessary to specialize Java programs using partial evaluation. Nevertheless, there are a few object-oriented language features that require further treatment. In addition, Java incorporates a number of language features that are not specifically object-oriented, such as reflection and exception-handling.

In this section, we give an overview of JSpec and describe the partial evaluation techniques specific to specializing Java programs. First, we list the primary JSpec features. Then, we discuss specialization of Java object-oriented language features, and last we discuss specialization of language features that do not have an object-oriented flavor. We do not investigate implementation issues here; Section 10.4 describes the technical details of the implementation.

10.3.1 Features

The JSpec partial evaluator takes as input a number of specialization class declarations and the Java code to specialize, optionally including C code for implementing native functionalities. The program code is analyzed according to the specialization class declarations and specialized according to concrete values provided by the user. The specialized code can be in the form of an AspectJ aspect, C source code, or binary executable code.

Input language

JSpec is controlled using the specialization class framework described in Chapter 6. Specialization classes offer a declarative means of expressing how a program should be specialized, and permit automatic reintegration of specialized code into the program (see Chapter 6 for details).

Java source code and bytecode are the primary input languages to the JSpec partial evaluator. Specifically, JSpec takes Java class files containing bytecode as input; Java source code is compiled into bytecode using a Java compiler. Although source code is usually required for the user to understand how a program should be specialized, a program that is to be specialized may make use of a library of routines available only in compiled form. Analysis of these external routines by the partial evaluator facilitates specialization, and with sufficient knowledge of their functionality, they can also be specialized.

The Java language makes it possible to interface Java code with native methods written in some other language, usually C, and defines a standard interface (JNI) for interaction between native code and Java code. Native methods are typically used to interface Java programs with operating system resources, such as the file system, network, and graphical interface, but may also be used to define computationally intensive routines. File system and network routines are known to be interesting targets for specialization [116, 134], as are computationally intensive routines used for scientific computations [14, 12] and image processing [70]. Thus, we consider Java native methods to be an interesting target for partial evaluation. JSpec uses C as its intermediate program representation; by supplying C functions that implement native functionality used by a Java program, these C functions can be specialized in the context of the Java program. Specialization simply generates a new set of specialized native functions to be used with the specialized Java program.

Analysis and specialization

In JSpec, type-polyvariance is controlled on a class-by-class basis; this feature allows the user to limit precision for classes that have uniform binding times. An alias analysis is used to determine the potential callees of a virtual dispatch; however, the alias analysis is computationally expensive. To circumvent this problem, a class-hierarchy analysis [50] is used before the alias analysis to simplify program control flow where possible. Specialization in JSpec is done using an efficient template-based generating extension that can generate source programs at compile time or binary executable programs at run time. The Harissa environment provides Java execution support during specialization.

Output language

Generation of residual programs as an AspectJ aspect isolates the specialized program part from the original generic program, and permits transparent plugging and unplugging of the specialized code. Weaving produces a standard Java program that can be compiled using standard Java tools and executed on any Java virtual machine.

Generation of residual programs as C source code allows the partial evaluator to optimize the specialized program beyond what is normally possible in Java. Operations built into the Java virtual machine can be optimized; array bounds checks with static bounds can be eliminated, and array store and class cast checks can be eliminated for static objects. Also, post-specialization inlining does not need to take Java visibility modifiers into account, since it is performed on C code.

Generation of residual programs directly as binary executable code allows specialization to be done at run time. Run-time specialization can take into account static values only available when the program is running. Analysis is done statically, and the specialized program is constructed out of pregenerated binary templates; this strategy makes run-time specialization highly efficient [120].

10.3.2 Specializing object-oriented language features

Given the techniques described in the two previous chapters, binding-time analysis is straightforward for the object-oriented parts of Java. The only new feature to take into account is interface calls, which are analyzed like virtual calls with abstract methods (i.e., by analyzing of the possible concrete methods); the set of possible callees can be predicted using type inference (as in Chapter 8) or alias analysis (as in Chapter 9).

With a few exceptions, the specialization phase is also straightforward for standard imperative and object-oriented Java features. Specifically, interface calls and constructors must be handled differently from other language features seen so far; final methods facilitate compiler optimization of the residual program; and interfaces are useful in conjunction with abstract methods defined in external classes.

Interface calls

Interface calls are similar to virtual calls that include abstract methods, but must be handled differently by the specializer. An error-generating method cannot be inserted into an interface, because interfaces cannot contain concrete methods (even just error-generating ones). To declare the method in the interface would require all classes that implement the interface to implement the specialized method. Rather, a new interface is introduced into the program. This interface declares only the name of the specialized method, and is only implemented by the classes included in the specialized interface call.

As an example, consider the program fragment of Figure 10.1a. The three classes `Sum`, `Mul`, and `Div` all implement the interface `IxI2I`, where a method `f` maps a pair of integers to a single integer. An object known only by its interface `IxI2I` is stored in the variable `fn`, and is used to compute a value. Suppose that the binding-time analysis infers that `fn` is dynamic and can reference objects of

```
interface IxI2I {
 int apply(int i,int j);
}
class Sum implements IxI2I {
 int apply(int i,int j) {
  return i+j;
 }
}
class Mul implements IxI2I {
 int apply(int i,int j) {
  return i*j;
 }
}
class Div implements IxI2I {
 int apply(int i,int j) {
  return i/j;
 }
}
...
 IxI2I fn = ...;
 res = fn.apply(s,d);
...
```

(a) Generic program

```
aspect Specialized {
 interface IxI2I_1 {
  int apply_0(int j);
 }
 private introduction Sum {
  implements IxI2I_1;
  int apply_0(int j) {
   return 1+j;
  }
 }
 private introduction Mul {
  implements IxI2I_1;
  int apply_0(int j) {
   return 1*j;
  }
 }
 ...
  IxI2I f = ...;
  res = ((IxI2I_1)f).apply_0(d);
 ...
}
```

(b) Specialized program

Figure 10.1: Speculative specialization of interface call.

type `Sum` or `Mult`, that the first argument `s` to `fn.apply` is static, and that the second argument `d` is dynamic. We can specialize this program fragment for `s` having value `1`, by specializing the possible callees of `fn.apply` for this value, and declaring the specialized method in a newly introduced interface named `IxI2I_1`; the result of specialization is shown in Figure 10.1b. New interfaces are added to classes using a special keyword `implements` in the `introduction` block for a class.

Constructors

A Java constructor contains object initialization code that can benefit from specialization. Constructors can essentially be specialized like ordinary methods, except that they cannot be introduced back into the class under a new name; multiple constructors are allowed only through overloading. In fact, JSpec uses overloading to introduce new constructors into a class. This strategy is illustrated in Figure 10.2a, with a complex number implementation that pre-computes and caches the absolute value of the number. Specialization generates a new constructor that directly makes use of the constructor parameters and values computed based on these parameters, as shown in Figure 10.2b. The specialized constructor is distinguished from the unspecialized constructor by passing an argument of type `_D_1`, where `_D_1` is a newly introduced dummy class. In the specialized version of the object instantiation expression, a `null` reference cast to the type `_D_1` is passed to the constructor. (The dummy argument

```
class Complex {
 float re, im, abs_cache;
 Complex(float r,float i) {
  set(r,i);
 }
 void set(float r,float i) {
  re=r; im=i;
  abs_cache=Math.sqrt(r*r+i*i);
 }
 float getR() { return re; }
 float getI() { return im; }
 float abs() { return abs_cache; }
}
...
 Complex c = new Complex(2.0,3.0);
...
```

(a) Generic program

```
aspect Specialized {
 introduction Complex {
  new(_D_1 dummy) {
   set_0();
  }
  void set_0() {
   re=2.0;im=3.0;
   abs_cache=6.0;
  }
 }
 class _D_1 {}
 ...
  Complex c = new Complex((_D_1)null);
 ...
}
```

(b) Specialized program

Figure 10.2: Specialization of constructors.

must be in an argument position where no other constructors take arguments of reference types, to avoid ambiguities when a constructor is passed a `null` reference not cast to a specific type.)

In Java, all constructors are required to call a constructor from the superclass, and a specialized constructor must obey this convention. If the superclass constructor is included in the program slice to specialize, a specialized superclass constructor that is called by the constructor will automatically have been generated by specialization. Conversely, if the superclass constructor is not included in the program slice to specialize, the default behavior is to not evaluate the constructor. Thus, a call to a generic constructor is residualized. The default behavior can be overridden by the user when the superclass has a default constructor; in this case, it assumed to be safe to implicitly call the the default constructor from the specialized code. The Java conventions for constructors are obeyed in all cases.

Final methods

Java has an explicit syntactic method modifier to declare non-virtual methods that can be called directly, named final methods. Although there is no syntactic difference between invoking a final and a non-final method, a compiler can trivially convert a virtual call to a final method into a direct call. Final methods are specialized like ordinary methods, and can be represented in the specialized program as final methods. Final methods can also be used to represent a direct call to a specialized method generated for a static-and-dynamic self object.

Abstract methods defined in external classes

Using the techniques described so far, specialization is not possible for a virtual dispatch that includes abstract methods defined in a class external to the program slice being specialized. An error-generating dummy method cannot be introduced into such an external class. In Java however, an interface can be introduced that only includes the concrete specialized methods generated, thus permitting specialization in this case. The virtual call is expressed like a specialized residual interface call, as shown in Figure 10.1.

10.3.3 Specializing non-object language features

There are essentially four "non-object" language features to take into account in partial evaluation for Java: reflection, exceptions, multi-threading, and dynamic loading. Our primary interest in Java is as an object-oriented language, and we consider full support for these language features in a complete partial evaluator to be future work. We now sketch how a partial evaluator for Java can be designed to cope with these language features.

Reflection

Java reflection enables run-time symbolic introspection and manipulation of objects. The Java reflection API can be dealt with as external code; reflection operations that are completely static can in this way be eliminated during specialization. For simplicity, we have chosen this approach in JSpec.

Most of the specialization opportunities in Java programs that use reflection cannot be specialized when reflection is considered an external operation. For example, reflection actions that manipulate dynamic data can often be statically determined. To better specialize programs that use reflection, statically determined reflection actions can be residualized as the equivalent Java code, as proposed by Braux and Noyé [25].

Exceptions

Exceptions introduce control flow that must be handled explicitly in a binding-time analysis. Program analysis in the presence of exceptions has been extensively studied in the context of Java [34, 36, 49, 104], but to the author's knowledge has never been investigated in the context of partial evaluation. We choose to only allow a very restrained use of exceptions: we disallow `catch` blocks in the program slice being specialized, residualize all `throw` statements that appear in the program, and consider it a specialization-time error if an exception is thrown during specialization. Hence, from the point of view of the partial evaluator, an exception throw has the same effect as terminating the program. Although `throw` statements are residualized, an exception can be thrown during specialization from an external method or, for example, by violating an array bounds check. A `finally` block implicitly catches and rethrows exceptions; for this reason we disallow `finally` blocks from the program slice being specialized.

Multi-threading

We take a pragmatic approach to dealing with multi-threading; we permit programs to be multi-threaded, but we only specialize a single thread of control. All synchronization and thread creation is considered dynamic, and will be residualized in the program. Statements within synchronized regions are speculatively specialized, which gives a simple and predictable semantics for specialization of multi-threaded programs. There are numerous cases where this specialization semantics would be undesirable. The development of a partial evaluation model that specializes multi-threaded programs in a useful way is considered future work.

Dynamic loading

Dynamic loading of classes into the program during execution is specified as part of the Java language. Partial evaluation relies on a static analysis of all parts of the program relevant to the program slice being specialized, and is thus highly sensitive to dynamic loading. Nevertheless, if dynamic loading is only used to select between a number of classes available during specialization, these classes can simply be included into the program as input to the partial evaluator. In the more general case, dynamic loading may however be used to load classes created after the program was deployed. These classes are naturally not part of the program slice being specialized, and their effect on the program slice must be described. To simplify applying partial evaluation to a program that uses dynamic loading, the program can be separated into individual components. Partial evaluation is applied internally to each component, and dynamic loading is used to load a complete specialized component.

10.4 JSpec Implementation

JSpec uses C as an intermediate representation during analysis and specialization, which permits direct reuse of the Tempo partial evaluator. The Harissa Java-to-C compiler maps the entire Java virtual machine into C for analysis and specialization, and the dedicated back-translator Assirah reconstructs a specialized Java program from the specialized C program. This architecture has permitted easy construction of a complete partial evaluator for the Java language, with only a few shortcomings.

In this section we first describe the architecture of JSpec and how each phase interacts with the next. Then, we describe Java-to-C translation, partial evaluation of the intermediate C code, and back-translation from C-to-Java. Last, we discuss a number of shortcomings in the current implementation. The C code shown in this section has been simplified and pretty-printed, but is otherwise similar to the intermediate C code used in JSpec.

10.4.1 Architecture

The JSpec partial evaluator is constructed as a combination of several separate components, as illustrated in Figure 10.3. The complete JSpec partial evaluator,

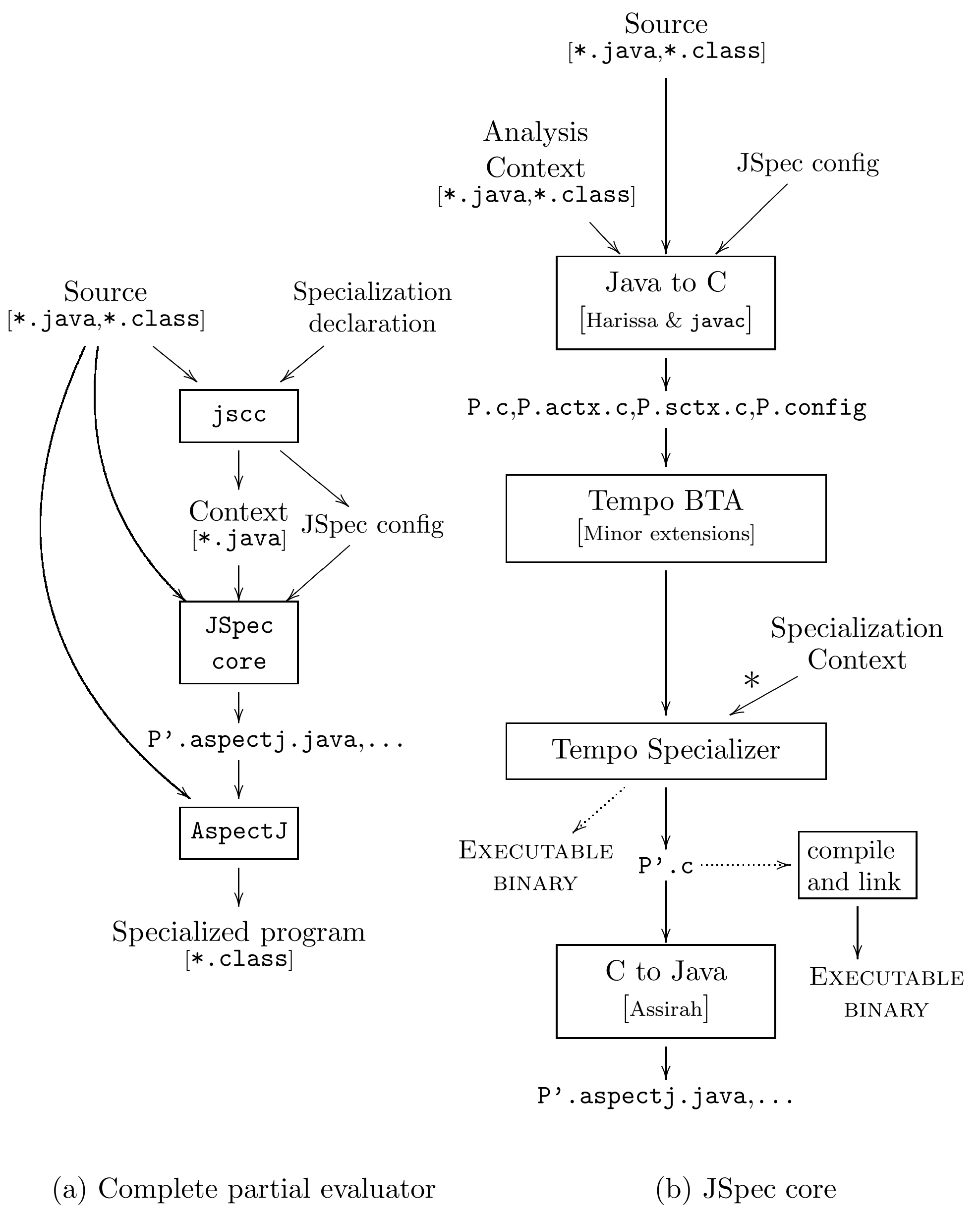


Figure 10.3: JSpec architecture.

shown in Figure 10.3a, takes as input a Java program to specialize, and a set of specialization classes declaring how it should be specialized. The specialization classes are processed by the specialization class compiler `jscc`, which produces a specialization context in Java code and a JSpec configuration file. These files are given as input to JSpec together with the source program, which produces a specialized program in the form of an AspectJ aspect; to produce a standard Java program, the specialized aspect is woven with the source program.

The JSpec core, shown in Figure 10.3b, consists of a Java-to-C converter, a C specializer, and a dedicated C-to-Java converter. The JSpec core takes as input a Java program to specialize, an abstract analysis context written in Java, and a configuration file that defines options to the specialization process. Conversion from Java to C is done using the Harissa Java-to-C compiler, which has been adapted to generate C code that is well suited for use with Tempo. Specialization of the generated C code is done using Tempo; for more information on how Tempo itself is structured into multiple phases, we refer to previous work [41]. Tempo analyzes the program once for the abstract context description, and subsequently specializes the program for a set of concrete values. Last, the specialized C code can either be used directly in the Harissa environment or translated back to Java source code encapsulated into an AspectJ aspect. Translation from C to Java is done using the *Assirah* C-to-Java converter. In general, arbitrary C code cannot be translated into equivalent Java. However, Assirah has been specifically designed to recognize C code generated by Harissa and specialized by Tempo, and will only translate such C code back into Java.

10.4.2 Java-to-C translation

Most parts of the Java language are easily mapped into equivalent C code. A Java class declaration can be represented as a pair of C structures: one structure for the class data and one structure for the object instance data. A virtual call can be expressed in C as an indirect call. Local variables, conditionals and loops map directly into their C counterparts. For other parts of the language, such as exceptions and multi-threading, the mapping is less obvious, but possible nonetheless in a low-level language like C; the C `longjmp` routine can be used to implement exceptions, and numerous thread libraries are available for C [117].

To facilitate optimization by the back-end C compiler, the Harissa compiler generates C code largely resembling what a human programmer could have written, as opposed to just using C as a "portable assembly language."[1] In particular, using C `struct` types to represent data rather than just pointers and numerical offsets was observed to yield superior performance. (Pointers and numerical offsets can be generated more easily than C structure declarations and named field access operations, and were for this reason used in the early versions of Harissa.) Using well-structured types makes it easier for a static analysis to trace the flow of information through the program. Even so, the standard translation provided by Harissa is optimized for compilation and

[1]This Harissa design decision was based on practical experiments with Gnu's `gcc` compiler and Sun's commercial `cc` compiler; the experiments are reported in Muller and Schultz's paper on Harissa [117].

execution efficiency, not for being analyzed and specialized by a partial evaluator and subsequently translated back into Java. For this reason, many minor modifications were made to the Harissa Java-to-C translation to enhance the results of the binding-time analysis and to facilitate subsequent specialization and back-translation.

The C program generated by Harissa implements Java subtyping through explicit casts between structures of compatible memory layout. Standard C semantics defines casts between pointers to structures of different type in terms of memory layout; when the memory layout is compatible, the values of the fields are preserved through the cast. The Tempo binding-time analysis supports type casts between structures in terms of projection of fields of the same name from one type to another; the memory layout is not taken into account. In the Harissa-generated C representation of an object, fields declared in a common superclass are given identical names; all other fields are given different names. Thus, when a field is accessed in the translated C program, only those aliases that are correctly typed in terms of the Java type system have fields named so that they affect the binding time of the field access. In doing so, information contained in the Java type system is also represented in the C program.

In the standard Harissa translation, virtual dispatching is implemented using an indirect function call over a table of virtual functions, represented as a C array of function pointers. However, Tempo relies on an alias analysis to determine the possible callees at an indirect function call site; this alias analysis does not discriminate between different indices of the same array, so all of the methods stored in the same virtual table (i.e., all the virtual methods of a single class) would be considered potential callees at a given call site. We observe that access to virtual tables always is done using indices that are known at compile-time, so the array of function pointers can be replaced by a structure that has a field for every entry in the virtual table. Doing so permits the alias analysis to precisely determine the possible set of callees at a given call site, since a field name now is used to indicate the callee. To make this scheme work with the treatment of type casts in the Tempo binding-time analysis, the name of a field holding a given virtual method is identical for all classes that override this method.

The most significant remaining changes to Harissa aim to express dynamic memory allocation in a Tempo-compatible format and to support program slicing for modular specialization. In Tempo, dynamic memory allocation is normally handled by requiring the user to manually convert each dynamic memory allocation operation into static memory allocation. Naturally, this conversion imposes a static constraint on the number of objects that can be allocated during specialization. Harissa has been modified to generate memory allocation operations that make use of statically pre-allocated memory; the user specifies the maximal heap size. To support program slicing at the Java class level, the C code that is generated for each class is separated into a concrete and an abstract description. When a class is not included in the program slice to be specialized, the abstract description serves to define its place in the class hierarchy, which is essential for correctly analyzing the program.

As an example of how Java code is translated into C code, we first give a complete example of translation of a Java class into C. Then, we show how a selection of small program fragments are translated into C; these program fragments will be used throughout the rest of the section to illustrate partial evaluation and back-translation into Java.

Example: complete Java class

The translation from Java to C for a Java class named `LL` is illustrated in Figure 10.4. The Java source code of the class `LL` is shown in Figure 10.4a, and its superclass `Object` is shown in Figure 10.4b. An object of class `LL` represents a single element in a linked list of integers. It has operations for computing a hashcode (on the object itself) and for computing the sum to a given depth of the integers in the linked list.

An instance of the class `LL` is represented by the C structure `_LL`, shown in Figure 10.4c. The first two fields of the `_LL` structure are common to all objects: there is a pointer to the class of the object, followed by a pointer to the virtual table (the virtual table is accessible from the class, but is for convenience pointed to from the object as well). Then follows a field for every Java field in the class, including any fields inherited from the superclass. Harissa translates the Java `int` type into the C type `TINT` defined by the Harissa environment rather than the standard C type `int`, since the precision of C base types varies with the platform, whereas Java base types have fixed size. The C type `TINT` is defined to have an appropriate precision that ensures correct Java semantics.

The translated representation of a Java class is shown in Figure 10.4d (all classes in the program are represented using this C structure type). The first two fields are the fields found in any object (in the Java language, a class is also an object). These are followed by fields that represent information internal to the class, such as the class name, the superclass, the virtual table associated with the class, and other class-level information. The virtual table of `Object` (shown in Figure 10.4e) contains a field for every virtual method in class `Object`. The virtual table of the class `LL` (shown in Figure 10.4f) contains the same fields as the virtual table for the class `Object` in a compatible layout. For example, the field `_hash` of the variable `VTable_Object_instance` points to the hash function of the class `Object`, whereas the field `_hash` of the variable `VTable_LL_instance` points to the hash function of the class `LL`. In addition, the virtual table of the class `LL` contains a field for every virtual method of class `LL` not found in class `Object`.

Last, Figure 10.4g shows the translation of the methods of class `LL`. The constructor (`_LL__init_`) takes as argument a structure of type `_LL`, and initializes each field (note that the default initialization provided by the `javac` compiler is used here when no value was explicitly assigned in the Java constructor). The method `hash` (`_LL_hash`) accesses the fields of the self object argument to compute a hash code. The `sum` method (`_LL_sum`) tests its argument (`_depth`) to determine whether to return zero or perform a recursive computation. The recursive computation is done using a virtual call, which is expressed as an indirect function call through the virtual table structure.

```
class LL extends Object {
 int e; LL next;
 LL(int e) { this.e=e; }
 long hash() { return e; }
 int sum(int depth) {
  return depth==0 ? 0
    : e+next.sum(depth-1);
 }
}
```

(a) LL class: linked list of integers

```
class Object {
 boolean equals() { ... }
 long hash() { ... }
 ...
}
```

(b) `Object` class (common superclass)

```
struct _LL {
 struct Class *class;
 struct VTable_LL *vt;
 /* object fields */
 TINT _e;
 struct _LL *_next;
};
struct Class _LL_class;
```

(c) LL object instance

```
struct Class {
 struct Class *class;
 struct VTable_Class *vt;
 /* internal class information */
 char *name;
 struct Class *superclass;
 void *class_vtable;
 ...
};
```

(d) `Class` object instance

```
struct VTable_Object {
 TBOOL (*_equals)(struct Object *);
 TLONG (*_hash)(struct Object *);
 ...
} VTable_Object_instance;
```

(e) Virtual table for `Object`

```
struct VTable_LL {
 /* methods from superclasses */
 TBOOL (*_equals)(struct _LL *);
 TLONG (*_hash)(struct _LL *);
 ...
 /* local virtual methods */
 TINT (*_sum)(struct _LL *,int);
} VTable_LL_instance;
```

(f) Virtual table for LL

```
void _LL__init_(struct _LL *this, int e) {
 this->_e = e; this->_next = NULL;
}
TLONG _LL_hash(struct _LL *this) { return this->_e; }
TINT _LL_sum(struct _LL *this,int _depth) {
 return _depth==0
        ? 0
        : this->_e + (*this->_next->vt->_sum)(this->_next,depth-1);
}
```

(g) Constructor and methods associated with LL

Figure 10.4: Translation of a Java class.

	Java	C
(a)	`x = new LL(1);`	`x = newObject_LL_87();` `_LL__init_(x,1);`
(b)	`x = y[i];`	`checkBounds(y,i,y->length);` `x = y->data[i];`
(c)	`x = ...` `y = x.hash();`	`x = ...` `y = (*x->vt->_hash)(x);`

Figure 10.5: Translation of selected Java statements.

Example: translation of code fragments from Java to C

Figure 10.5 illustrates translation from Java to C of three basic Java operations: object allocation (10.5a), array element lookup (10.5b), and virtual dispatch (10.5c). These three statements will be used in the subsequent sections to demonstrate the partial evaluation and back-translation phases. Java object instantiation is decomposed into allocation of an object using a specific memory allocator generated by the translation followed by a constructor invocation on the new object and the original arguments. An array is represented as a C structure with two fields, `length` that holds the length of the Java array, and `data` that is a C array holding the contents of the Java array. An array element lookup is decomposed into an array bounds check and the actual access to the array. In Java, array indexing and storing are checked at run time, and type casts likewise; the translation from Java to C makes these operations explicit. Finally, a virtual call is translated into an indirect function call, as explained earlier in this section (we suppose that the statement "`x = ...`" shown in the figure evaluates to an object either of type `Object` or of type `LL`).

10.4.3 C partial evaluation

Partial evaluation of the Harissa-generated C program is accomplished using Tempo. Tempo specializes the entire C language, although with restrictions on pointer usage. The Tempo binding-time analysis is method polyvariant, type polyvariant, flow sensitive and use sensitive. Specialization is done using a generating extension. Partial evaluation for C is essentially done according to the partial evaluation principles presented so far, extended to deal with pointers and side-effects.

The only significant modification that was made to Tempo to facilitate specialization of Harissa-generated code is the implementation of type polyvariance in the binding-time analysis. This analysis feature can often be useful in C, but is essential when specializing realistic Java programs. Apart from type-polyvariance, numerous minor enhancements have been made to Tempo that are useful both for normal C programs and Harissa-generated programs. To a large extent, these features have been added to deal with the additional complexity of the Harissa-generated C programs over standard C programs.

Example: partial evaluation of C code fragments

Partial evaluation of the C program fragments of Figure 10.5 is done as follows. Object allocation is treated as an external operation, that can either be eliminated by specialization or residualized in the specialized program. If an object is used statically throughout the program, it will only be allocated during specialization. If an object is static-and-dynamic, it is allocated and used during specialization; the object allocation operation will be residualized in the specialized program. Lastly, if an object is dynamic, the object allocation is simply residualized into the specialized program. The associated constructor invocation is simply specialized like a method invocation, as described in Section 10.3.

Figure 10.6 illustrates how the array element lookup and virtual dispatch shown in Figure 10.5 are specialized by Tempo. Partial evaluation of these code fragments relies on how Tempo specializes external functions; basically, an external function can be evaluated during specialization when all of its arguments are static.

Specialization of array element lookup is shown in Figure 10.6a. The `checkBounds` function used to express array element lookup is an external function; its arguments (the array, its length, and the index to look up) are all static when and only when the array bounds check can be performed during specialization. An array bounds check that is performed during specialization will signal a specialization error if the index is out of range. If the index is within range, the `checkBounds` function is removed by specialization, and lookup of a static array element can be performed during specialization. If the array element is static-and-dynamic or dynamic, the array lookup is residualized without a bounds check.

Specialization of virtual dispatching is shown in Figure 10.6b. Tempo does not directly support indirect calls through function pointers. Rather, information from the alias analysis is used to transform each indirect function-pointer call into an explicit conditional that tests the function pointer to decide which function to invoke using a direct call. The conditional is placed in a separate function named `apply` (shown in Figure 10.6c). The conditional can be reduced when the function pointer is statically known; this situation occurs when the pointer to the receiver object is statically known. In terms of partial evaluation for Java, when an object reference is statically known, virtual dispatches over the object can be reduced. The self object is removed by specialization when the object reference is static, and residualized when the object reference is static-and-dynamic. If the receiver object is dynamic then the function pointer is dynamic, and each potential receiver method is specialized speculatively. All cases are illustrated for the `apply` function in Figure 10.6c. If during specialization a virtual dispatch is done over an object that was not given as input to the binding-time analysis, then none of the test cases match. This situation only occurs when the context provided for analysis does not match the concrete data supplied for specialization. In this case, the function `_DispatchError` is called; it signals a specialization error. The `_DispatchError` function is passed the function pointer as an argument, which ensures that it is not called due

(a)

Generic C

```
checkBounds(y,i,y->length);
x = y->data[i];
```

Specialized C

`i`, `y.length` are static ($i = 0$) ⇒

```
x = y->data[0];
```

`i`, `y.length` are dynamic ⇒

```
checkBounds(y,i,y->length);
x = y->data[i];
```

(b)

Input C

```
x=...
y=(*x->vt->_hash)(x);
```

Tempo transformed C

```
x=...
y=apply(x->vt->_hash,x);
```

`x` is static (`x` is an `LL`) ⇒

```
y=apply_0();
```

`x` is stat-and-dyn (`x` is an `LL`) ⇒

```
x=...
y=apply_0(x);
```

`x` is dynamic ⇒

```
x=...
y=apply_0(x->vt->_hash,x);
```

(c)

Generic apply function

```
TLONG apply(
 TLONG (*f)(struct Object*),
 struct Object *v0
) {
 if(f==_Object_hash)
  return _Object_hash(v0);
 else if(f==_LL_hash)
  return _LL_hash(v0);
 else
  return _DispatchError(f);
}
```

Specialized apply functions

`x` is static (`x` is an `LL`) ⇒

```
TLONG apply_0() {
 return _LL_hash_0();
}
```

`x` is stat-and-dyn (`x` is an `LL`) ⇒

```
TLONG apply_0(
 struct Object *v0
) {
 return _LL_hash_0(v0);
}
```

`x` is dynamic ⇒

```
TLONG apply(
 TLONG (*f)(struct Object*),
 struct Object *v0
) {
 if(f==_Object_hash)
  return _Object_hash_0(v0);
 else if(f==_LL_hash)
  return _LL_hash_0(v0);
 else
  return _DispatchError(f);
}
```

Figure 10.6: Partial evaluation of translated statements.

to speculative specialization (the branches of the conditional are speculatively specialized when the function pointer is dynamic).

C code output

When JSpec is used to generate specialized programs as C code or binary executable code, no further translation is performed. The specialized program generated by Tempo is executable in the Harissa environment, after the static memory allocations have been converted back into dynamic memory allocations. Naturally, Java virtual machine operations, such as array bounds and store checks or type cast checks that have been eliminated by specialization are not performed in the specialized program.

Specialized virtual dispatches are not converted back into an indirect call, but remain as a sequence of conditionals in the specialized C code. An optimizing compiler for an object-oriented language may use the same representation for a virtual dispatch [69, 76]; this optimization is referred to as receiver-prediction. However, receiver-prediction is only used when the number of callees is low, whereas specialized virtual dispatches are always represented as conditionals in specialized C programs. Generating indirect function call would require modification of the virtual tables of the program; such a modification would be non-trivial after translation from Java to C, so we choose to leave this inefficiency as a subject for future work.

10.4.4 C-to-Java translation

The Assirah C-to-Java back-translator has been constructed to recognize the patterns of C code produced by partial evaluation of Harissa-generated Java programs. Rather than to attempt to analyze the C structure declarations that describe the class structure of the program, Assirah relies on auxiliary information supplied by Harissa that describes each C function, structure declaration and global variable in the specialized program in terms of its meaning in the original Java program.

Example: translation of code fragments from C to Java

Specialization of Java virtual machine operations, such as the array bounds check shown in Figure 10.6, cannot be translated into Java. For this reason, a residualized array element access in C is simply translated back into Java code that performs the array element access, regardless of whether the `boundsCheck` operation has been removed by specialization or not.

Figure 10.7 illustrates how the specialized virtual dispatch shown in Figures 10.6b and 10.6c is translated back into Java. In the first case, shown in Figure 10.7a, the conditional over the function pointer and the self object has been reduced away, and the specialized function is back-translated as a static method that is directly invoked. The second case shown in Figure 10.7b is similar, except that the specialized function is back-translated as a final method that is invoked using a type cast. In the third case, shown in Figure 10.7c,

```
     Specialized C                          | Specialized Java
     ---------------------------------------+--------------------------------------
                                            | aspect Specialized {
                                            |
     y=apply_0();                           |     ...
     ...                                    |     y=LL.hash_0();
                                            |     ...
(a)  TLONG apply_0() {                      |
      return _LL_hash_0();                  |  private introduction LL {
     }                                      |   static long hash_0() { ... }
                                            |  }
                                            | }
     ---------------------------------------+--------------------------------------
                                            | aspect Specialized {
     x=...;                                 |
     y=apply_0(x);                          |     ...
     ...                                    |     x=...;
                                            |     y=((LL)x).hash_0();
(b)  TLONG apply_0(                         |     ...
      struct Object *v0                     |
     ) {                                    |  private introduction LL {
      return _LL_hash_0(v0);                |   final long hash_0() { ... }
     }                                      |  }
                                            | }
     ---------------------------------------+--------------------------------------
     x=...;                                 | aspect Specialized {
     y=apply_0(x->vt->_hash,x);             |
     ...                                    |     ...
                                            |     x=...;
     TLONG apply_0(                         |     y=x.hash_0();
      TLONG (*f)(struct Object *),          |     ...
      struct Object *v0                     |
(c)  ) {                                    |  private introduction Object {
      if(f==_Object_hash)                   |   long hash_0() { ... }
       return _Object_hash_0(v0);           |  }
      else if(f==_LL_hash)                  |  private introduction LL {
       return _LL_hash_0(v0);               |   long hash_0() { ... }
      else                                  |  }
       return _DispatchError(f);            | }
     }                                      |
```

Figure 10.7: Back-translation of specialized statements into Java.

the conditional still remains; each possible callee is back-translated as an ordinary method that is invoked using an ordinary method invocation. In all cases the specialized methods are encapsulated into an AspectJ aspect. After back-translation, weaving inserts the specialized methods into the classes indicated by each `introduction` block.

As explained in the previous chapters, a number of special cases must be taken into account when expressing a specialized virtual dispatch in Java. For example, the set of possible callees determined by the alias analysis for a residualized dynamic virtual dispatch might not include a common superclass (solution: introduce an error-generating dummy method in the common superclass), or it might include a method from an external class where no new methods can be added (solution: residualize unspecialized virtual dispatch). In addition, there are three complications that are specific to our `apply`-function representation. First, in a residualized dynamic virtual dispatch to methods that return a residualizable value, speculative specialization of a branch (a method invocation) may have caused the branch to be replaced by the result of invoking the method, i.e., a constant. In this case, it is the function-pointer test on the branch returning the value that indicates which callee has been specialized into a constant. Second, the specialized potential callees may have different numbers of formal parameters. This situation can occur in JSpec when an object argument is used in a dynamic context in some but not all of the callees; binding times are flow-sensitive, so a given argument may be static-and-dynamic for some callees and static for others. The static arguments are removed by the specializer, but must be reintroduced by the back-translation to have a uniform method signature. To reintroduce arguments, the back-translator must keep track of the names of method formal parameters before and after specialization. Third, JSpec performs inlining optimization after specialization but before back-translation. To allow back-translation, inlining must preserve the structure of the `apply` functions generated by Tempo.

10.4.5 Shortcomings of the current implementation

The analysis phase is the most critical part of JSpec; it is the analysis phase that determines how much specialization can be done based on the static values of the specialization context. However, it remains difficult to structure the program and specify the appropriate specialization context to obtain the desired binding-time result. The techniques presented in Part II of this document simplify the use of partial evaluation, but are not sufficient. Compared to partial evaluation for functional and imperative languages, the extensive use of side-effected heap-allocated data structures (objects) and the often intricate control flow (because of virtual dispatching) are complicating factors. The development of binding-time analyses that are easier to use are considered future work.

The current implementation of JSpec lacks a number of features; although simple to implement in principle, some of these features require a significant amount of work to implement in practice.

- The specialization-class compiler has a number of limitations, as described in Chapter 6. In addition, the current version of the compiler cannot generate adequate Java code for expressing the specialization context; although specialization classes have been written for the experiments reported in this document, the specialization context has been written manually. We expect to release a complete implementation soon.

- The degree of Java support is limited by the Java support in Harissa. Currently, only the features of JDK 1.0.2 are fully supported, with partial support for JDK 1.1.

- The JSpec alias analysis is monovariant; some programs that we specialize with JSpec have been very carefully written to ensure appropriate binding times, others have been modified manually. The manual modifications were done according to knowledge of what binding times were needed at critical program points, and essentially consist of duplicating code to overcome alias analysis imprecision or forcing specific binding times using special Tempo features. We are currently contemplating how the alias analysis can be extended to be polyvariant.

- JSpec does not support abstract descriptions of external code for analysis purposes. External code can be visible to the analysis, which is essential for partial evaluation of large programs, but it is only the actual code that can be visible to the analysis; there is no support for supplying an abstract description of external code different from the actual code. External code is always available in the form of bytecode, but it is desirable that a version of the external code that reflects the specialization intent can be specified.

- There is no garbage collection during specialization. Normally, few objects are allocated during specialization, but garbage collection can be essential for specializing certain programs.

- There is no object inlining nor common subexpression elimination performed during post-specialization optimization.

- Java source code is first translated to bytecode before being translated to C, which implies the loss of local variable names and intraprocedural program structure. The use of a direct compiler from Java to C would improve the appearance of the specialized program, which would facilitate inspecting the result of specialization.

- There is currently only a limited support for generating specialized C source or binary executable code, which complicates the use of this feature. Specifically, the set of files generated by JSpec must be manually assembled and integrated into the generic program.

In addition to these implementation limitations, JSpec has a conceptual shortcoming that cannot easily be remedied. Run-time specialization can only generate binary code for execution within the Harissa environment; Java bytecode cannot be dynamically generated. We discuss the development of partial evaluation principles that allow run-time specialization of Java bytecode in Chapter 13.

10.5 Software Engineering Experiences

In the design of JSpec, we have chosen to re-use the Tempo partial evaluator for C as the partial evaluation engine. The choice was based on the exploratory nature of this work: to know the features needed from a partial evaluator to specialize large Java programs, it was first necessary to experiment with partial evaluation for Java. To use an existing partial evaluator is initially less work than writing a new one from scratch. Nonetheless, with the completion of JSpec, we can evaluate to what extent this approach was successful.

In this section, we consider whether we in the development of a new partial evaluator for Java still would base the implementation on a C partial evaluator such as Tempo, or would prefer a partial evaluator written for Java from scratch. We first discuss the use of C as an intermediate language in JSpec, and then propose a revised design based on our experience.

10.5.1 C as an intermediate language

The use of C as an intermediate language offers some immediate advantages: the partial evaluator can trivially produce specialized programs in C, and there is a large degree of implementation sharing between the C partial evaluator and the Java partial evaluator. Java virtual machine operations such as array bounds check and class cast check can be specialized, as can native methods.

From an intraprocedural point of view, partial evaluation of Java programs is basically like partial evaluation of C programs. Hence, many of the same features are needed in the partial evaluator, and using the same implementation reduces initial development work and facilitates maintenance. From an interprocedural point of view, the low-level nature of C makes it possible to translate Java-specific features, such as the class hierarchy and virtual dispatching, into an intermediate C representation.

In general, translating from Java to C for specialization imposes an intrinsic restriction, in that specialization must take place in the environment provided by the translation. In the case of JSpec, evaluation of static Java code must take place in the Harissa environment, no other Java virtual machine can be used. Thus, specialization is limited to the Java version provided by the Harissa environment, and can only take place on a platform supported by Harissa. In contrast, had specialization taken place in a generic Java environment, any Java virtual machine on any platform could have been used for specialization.

Translation into C has an additional disadvantage: specialization-time errors (such as dereferencing a null reference during specialization) are generated and must be handled at the C level, requiring intimate knowledge of the Java-to-C translation on the part of the user to correctly respond to specialization-time errors.

As illustrated in Section 10.4.2, the translation of Java into the intermediate C representation can be complicated, in particular for features such as the class hierarchy and virtual dispatching. In fact, the translation encodes the Java class hierarchy into C by using pointers between C structures. This encoding of key Java features into C introduces an additional layer in the representation of the program, which is a major overhead during analysis. In effect, the binding-time analysis interprets the C encoding to obtain the desired Java-level results; analysis of medium-size Java programs can consume several hours of computation. In addition, Java-level analysis and transformation features in the partial evaluator core cannot be directly expressed in terms of Java syntax, but must be indirectly expressed in terms of the encoded C syntax. Currently, such complications have primarily manifested themselves for the back-end translator from C to Java, for example for virtual calls. Nonetheless, as JSpec is further developed and more features are added, it seems likely that similar complications may arise in other JSpec phases.

As an example of where Java-level features are needed, consider external methods. For external methods, the default behavior given by Tempo to external functions during analysis is impractical for Java programs. As described in Chapter 9, a useful default behavior for external methods would be that they depend on the fields of the self object, and return a dynamic object. Such a behavior could be implemented in the Java-to-C translation by automatically generating abstract descriptions of all external methods. However, doing so would add a massive overhead to the analysis due to the additional amount of program code that would have to be analyzed. For example, one such function would have to be generated and potentially analyzed for every method in the standard JDK library. It only seems realistic to implement the desired behavior directly in the Tempo binding-time analysis, which breaks with the hitherto approach of not adding Java-specific functionality to Tempo.

In general, partial evaluation of Harissa-generated programs is more complicated than partial evaluation of standard C programs, and extended support for dealing with C features heavily exploited in the Harissa-based translation may be needed in Tempo. As an example, consider memory management; Tempo currently only supports statically-allocated memory, and Harissa translates Java dynamic memory management into static memory management. The translation relies on the user to specify a maximal heap size to use during specialization. If the heap size is exceeded specialization fails, and the heap size must be increased. However, this modification changes the data-structures of the translated C program, so the program must be retranslated and reanalyzed (a lengthy process). The size of the heap is constrained since the entire heap currently must be saved every time speculative specialization is performed. Direct support in Tempo for dynamic memory allocation would simplify specializing Java programs using JSpec.

10.5.2 Proposal for a revised design

The use of Tempo as the partial evaluation engine of JSpec has proven largely successful. There is a large degree of implementation-sharing made possible by the strong similarities between partial evaluation for C and partial evaluation for Java. We conclude that the basic premise of our JSpec design, to re-use Tempo, was a good idea. However, to simplify and streamline the JSpec implementation, it seems worthwhile to add direct support in Tempo for certain core Java features.

Direct support in Tempo for virtual dispatching, dynamic memory allocation, external Java methods that use the fields of the self object and return objects, and a class hierarchy would optimize the overall implementation and facilitate specializing Java programs. The complications in specializing virtual dispatches could be reduced by representing them directly in Tempo; restrictions on how virtual dispatches should be specialized can be expressed directly rather than enforced by a syntactic analysis in the back-translator. Direct support for dynamic memory allocation would alleviate the need for a statically specified upper bound on the size of the specialization heap. Java-specific behavior for external methods would reduce the amount of information that has to be specified by the user to specialize a program. Knowledge about the Java class hierarchy could be used to refine the result of the alias analysis, and to improve the default behavior for the return values of external methods.

10.6 Summary

We have extended the basic partial evaluation principles presented in the previous chapters to encompass the object-oriented features of the Java language, while taking a more pragmatic approach to dealing with some of the more exotic non-object features. Based on these extended principles, we have implemented a complete partial evaluator for Java, named JSpec. This partial evaluator is implemented using a translation-based approach with C as an intermediate language. This strategy has permitted direct re-use of existing technology.

JSpec is encumbered by the representational overhead introduced by the translation from Java to C and lacks some Java-specific analysis features. Nonetheless, JSpec is the first fully functional partial evaluator for a realistic object-oriented language such as Java. JSpec has made it possible to conduct practical experiments to determine those features that are essential to partial evaluation for object-oriented languages, and the effectiveness of JSpec partial evaluation on the benchmark programs of the next chapter illustrates the viability of the partial evaluation principles presented in this document.

Chapter 11

Experimental Study

11.1 Introduction

There has in the development of computer science been a continuous effort to improve the execution speed of programs; improvements both in hardware and software technology have made modern-day computers billions of times faster than the first computers that were built. However, the complexity of software continues to increase, and the optimization of program speed remains an active area of research; our approach to increasing program speed is partial evaluation.

Partial evaluation can give massive increases in program speed under the right circumstances [84]. However, there is almost no existing work that examines the impact of the optimizations performed by partial evaluation on object-oriented programs. Furthermore, we target the Java language; a Java program can be interpreted or compiled before execution, and can also be executed on numerous different systems that perform run-time optimizations. Very little is known about the speedup due to partial evaluation when the specialized program is executed on a system that perform run-time optimizations. A systematic study of the benefits due to partial evaluation of object-oriented programs across different compilation systems is needed.

The purpose of this chapter is to present an experimental study of how JSpec optimizes Java programs. The large variety in execution environments causes an explosion in the number of factors that affect the performance improvement from partial evaluation. However, each machine architecture and execution platform gives rise to a general behavior, and we can characterize the speedup benefits due to partial evaluation. We find that existing compiler technology is complementary to partial evaluation, and that an aggressive optimizing compiler is essential for attaining the full benefits due to this technique.

Chapter overview: First, Section 11.2 discusses general issues in the interaction between compiler optimizations and partial evaluation, and Section 11.3 describes various models for Java execution. Then, Section 11.4 gives an overview of the benchmarks that we will use for testing the benefit due to partial evaluation, and Section 11.5 presents the actual results. Last, Section 11.6 presents an assessment of the experimental results, and Section 11.7 presents a summary.

11.2 Compiler Optimizations vs. Partial Evaluation

Optimizing compilers often integrate optimizations akin to the transformations performed by partial evaluators. Since a partial evaluator requires more involvement from the user than a compiler, the application of partial evaluation should normally be restricted to the elimination of overheads that cannot be removed by a compiler. In this section, we first discuss the partial overlap of compiler optimization and partial evaluation in the case of object-oriented languages, and then describe a number of issues to take into account when using partial evaluation to optimize programs.

11.2.1 Object-oriented compiler optimizations

There is a fundamental difference between compilers and partial evaluators. On the one hand, a compiler is normally required to terminate, which inevitably forces it to be conservative when optimizing a program (it cannot solve the halting problem). On the other hand, a partial evaluator is normally not required to terminate, and can thus potentially perform more optimizations than a compiler. This difference manifests itself in the way compilers and partial evaluators are implemented. A compiler typically uses a symbolic analysis to reduce computations within a fixed time bound, whereas a partial evaluator typically uses standard evaluation of selected program fragments. The compiler does not propagate known information through those parts of the program that cannot be determined to be safe to reduce; the amount of computation removed by the compiler is limited by the precision of its analyses. In contrast, partial evaluation propagates known information through those parts of the program that can be evaluated; the amount of computation removed by the partial evaluator is limited only by the time spent to specialize the program.

For imperative languages, optimizing compilers are usually very conservative with regards to the amount of computations that they reduce; they rely on a combination of standard intraprocedural optimizations (e.g., copy propagation, constant propagation, ...) and low-level optimizations to produce efficient code. For object-oriented languages, the computation that takes place to decide the receiver of a virtual dispatch is in many cases sufficiently simple that aggressive interprocedural optimization becomes feasible. By propagating information about the identity of an object throughout the program, virtual dispatches can be reduced. Ensuing optimizations, such as inlining and standard intraprocedural optimizations, allow efficient code to be generated. However, the degree of information propagation is limited by the extent to which the symbolic analyses of the compiler can propagate information throughout the program; the result is that optimization tends to be of a local nature rather than a global nature.

Given information about how the objects of a program are composed together, partial evaluation propagates this information globally throughout the program, as long as the information is manipulated under static control. The information about object composition is used to reduce any statically controlled virtual dispatches; it is obtained either from the user or by evaluation of the initialization code of the program. In effect, partial evaluation permits code to be generated that is dedicated to a specific configuration of the program objects.

From these considerations we conclude that a compiler for an object-oriented language potentially can optimize a program as well as a partial evaluator when the interaction that takes place between the objects of the program is detectable using a static analysis or profile information. Specifically, when the degree of polymorphism in a program is limited, these techniques suffice to optimize most virtual dispatches, since there only is a single callee for each call site in the program. On the contrary, when each instance of an object is parameterized by objects of different classes, virtual dispatches become hard to predict. In this case, partial evaluation can optimize the program based on information on how each object is parameterized by other objects.

11.2.2 Specializing for program speed

In many scenarios, and indeed in most of the benchmarks presented in this chapter, partial evaluation can be said to trade code size for execution speed. The specialized program is often larger than the generic program, but is simpler to optimize, and as a consequence, faster to execute. But not all compilers can cope with the kinds of programs generated by a partial evaluator. Furthermore, partial evaluation might not always provide similar benefits across different computer architectures.

As an example of how the compiler impacts the efficiency of the specialized program, consider a complex program that has been simplified into a single large method using partial evaluation. The control flow is straightforward, and thus most compiler optimizations are simple to apply. However, some compilers might consider that intraprocedural optimizations such as copy propagation and dead-code elimination are too expensive to perform for such a large procedure, and likewise for essential, low-level optimizations such as register allocation and instruction scheduling. This issue is especially relevant when optimizations take place at run time, since strong restrictions are imposed on the amount of computation that can be used to optimize each method of a program.

As an example of how the computer architecture can effect the efficiency of the specialized program, consider a dynamically controlled loop enclosing a statically controlled loop. Unrolling the inner loop and generating specialized versions of the loop body can increase the size of the body of the outer loop. If the size of the specialized loop exceeds the size of the instruction cache, performance can be heavily impaired, even though the program has been specialized well.

11.3 Java Execution Models

The Java class file format was originally designed for interpretation on an embedded system. Since then many alternative Java compilation strategies have been devised, each best suited for specific scenarios. This section gives an overview of the various standard Java execution strategies, and describes their respective advantages and disadvantages.

Interpretation

Java bytecode is designed to execute on an abstract stack machine with object operations, which can be straightforwardly implemented as an interpreter. In general, interpreters are simple to implement and easily portable at the cost of execution speed. The dynamic nature of Java (e.g. dynamic loading and reflection) is easily captured in an interpreter, and the compactness of the Java bytecode format makes it well suited for interpretation on an embedded system.

Off-line compilation

The most common way of executing a program is to compile it into native code before execution, using an off-line compiler. Off-line compilation has the advantage that complex and time-consuming analyses can be used to guide optimizations, although optimizations must necessarily be conservative to avoid unnecessary optimization of rarely used program parts (unless somehow guided by profile information — see below). Off-line compilation for Java works best when the entire program is available at compilation time since whole-program optimizations can be performed, but this is not a requirement.

JIT compilation

A JIT (just-in-time) compiler differs from a "classical" off-line compiler, in that the code is compiled at execution time, and only when needed. The basic scheme is to compile a method when it is called the first time, pausing execution while doing so. JIT compilers can perform many whole-program optimizations in the presence of dynamic loading (since the program simply can be recompiled after dynamic loading), and can optimize for the specific processor model on which a program is executing. On the down side, the optimizations that can be performed at run time must be computationally inexpensive since they consume processor time that otherwise would have been spent executing the program. In addition, JIT compilers are highly platform dependent, since they must generate native machine code at run time for a specific processor.

Adaptive compilation

An adaptive compiler essentially works like a JIT, except that it only optimizes those parts of the code that execute most often; rarely executed program parts are either interpreted (the approach taken by HotSpot [150]) or quickly compiled (the approach taken by Jalapeño [4]). Furthermore, an adaptive compiler will normally use profile information to determine critical program parts that are further optimized using aggressive optimizations (e.g. selective argument specialization [48] or similar techniques). Compared to an off-line compiler, performing optimizations at run time has the obvious advantage that more information is present when the program is running, including computed values and dynamically gathered profile information. Compared to a JIT, less time is spent optimizing rarely executed program parts, and more time is spent optimizing the critical program parts. The advantages of adaptive compilation come at the expense of a certain startup time before the program is optimized; moreover, performing optimization adaptively temporarily slows down the program.

Hybrid approaches

A compilation strategy need not be constrained to one of these four categories. For example, Harissa is an off-line compiler that integrates an interpreter designed to run dynamically loaded code [117]. The Vortex compiler implements highly aggressive optimizations such as customization and selective argument specialization, but in an off-line approach; its static, global analyses are complemented by an automated profiling system that guides optimizations [49]. Alternatively, the on-line and off-line strategies can be made complementary: a compilation process can consist of running the unoptimized code while another process does aggressive compilation in the background, as shown by Plezbert and Cytron [131]. Once the optimized code is available, the unoptimized code is replaced with the optimized one; this scheme is referred to as *continuous compilation.*

11.4 Description of Benchmark Programs

To properly explore the execution speed advantages due to partial evaluation, we have aimed for a wide selection of benchmark programs. This section describes our test programs, grouped by the primary kind of specialization opportunity that is found in each program. We group specialization opportunities into imperative, object-oriented, and mixed. With the exception of the programs with imperative specialization opportunities, these programs are written in a generic programming style; this design decision is an advantage in terms of software engineering, but is a disadvantage in terms of performance. For each benchmark program, we give a brief description of its functionality, describe what program input data is considered static, and finally describe the input for which the program is specialized in the experiment reported in the next section. All benchmark programs are computationally intensive; no large amount of I/O is performed, no large amount of memory is allocated, and only a single thread of execution is used.

In all of the benchmark programs, we have avoided the use of access modifiers, to enable uninhibited inlining as a post-specialization optimization; although unrealistic, this choice allows us to measure the benefit due to the inlining that can be performed by JSpec.

11.4.1 Imperative opportunities

Some object-oriented programs are primarily imperative in their nature, although they may benefit from object-oriented constructs to provide structuring or data encapsulation. The benchmark programs that exhibit imperative opportunities for specialization could have been written in an imperative language and then specialized using a partial evaluator for this language. However, little is known about the efficiency of imperative Java programs after partial evaluation, and so we consider it interesting to include such programs in our benchmark suite.

FFT: The Fast-Fourier Transform benchmark taken from the Java Grande benchmark suite [60]. The radix size is static, and the data being transformed is dynamic. Specialization is done for three different radix sizes, 16, 32, and 64.

Power-1: A standard version of the power function, written using a loop and with a fixed operator (multiplication) and neutral value. The exponent is static, and the base value being raised is dynamic. Specialization is done for an exponent value of 16.

With the possible exception of off-line compilers, most Java compilers are not tuned to optimizing imperative code. Because of the lack of compiler optimizations, we expect that the improvements due to partial evaluation will be less noticeable, and the speedup due to partial evaluation will be smaller than for similar programs in an imperative language.

11.4.2 Object-oriented opportunities

The adaptive behavior of some object-oriented programs is completely controlled through object-oriented mechanisms; such programs can be said to provide pure object-oriented specialization opportunities. Simplification of virtual dispatches without taking imperative features such as loops, conditionals, and other computations into account is sufficient for specializing these programs.

Builder: An extended version of the builder example from Chapter 7; a builder design pattern is used to construct matrices either with a dense or sparse representation, which are subsequently multiplied. The choice of builder pattern is static, and the matrix dimensions and contents are dynamic.

Bridge: An extended version of the bridge example from Chapter 7; the complex numbers from the example are used in the computation of a Mandelbrot fractal. The bridge coupling is static, and the actual complex numbers manipulated are dynamic.

Iterator: An extended version of the iterator example from Chapter 4; the iterator pattern is used in the implementation of member and set intersection operations over an underlying primitive data structure. The choice of underlying data structure is static, and the data being manipulated is dynamic.

We expect to have significant speedups due to partial evaluation when using off-line compilers, since partial evaluation can simplify the program beyond what most standard optimization techniques can accomplish. As for JIT and adaptive compilers, they can, at least to some extent, perform the same kinds of transformations that a partial evaluator performs; as a result, we expect the speedup due to partial evaluation to be lower than for off-line compilers.

11.4.3 Mixed opportunities

In many object-oriented programs, the adaptive behavior is controlled by a mix of object-oriented mechanisms and imperative constructs. In the programs exhibiting "mixed" specialization opportunities, specializing the program to a fixed input requires treating a mix of object operations and imperative computations.

ArithInt: The arithmetic expression interpreter from Chapter 5, used to compute the maximal value of a function. The arithmetic expression is static, and the contents of the environment is dynamic. Specialization is done for a function mapping integer planar coordinates into an integer value.

ChkPt: The checkpointing example presented by Lawall and Muller [97]: a generic checkpointing routine for a binding-time analysis implementation is specialized to object composition properties specific to each phase of the binding-time analysis. Specialization and benchmarking are done for the binding-time analysis phase only. This example is further described in Chapter 12.

Fold: The fold example from Chapter 8 (a binary operator is folded over a list). In experiment LS, the list contents are static, and the operator and initial value are dynamic. In experiment OP, the operator and initial value are static, and the list contents are dynamic. Specialization is done for a list of length 50 and a multiplication operator.

IP: The image processing example from Chapter 1. Specialization is done for blurring convolution filters of size 3×3 and 5×5.

Pipe: Simple mathematical functions composed together to form a pipe, applied to a single input value. The function composition is static, and the value input to the function pipe is dynamic.

Power-2: The power example from Chapter 5, rewritten with a standard Java `for` loop. The exponent, operator and neutral values are static, and the base value is dynamic. Specialization is done for a multiplication operator and an exponent of 16 (making the end result similar to the specialization of Power-1).

Strategy: An extended version of the strategy example from Chapter 7; a number of single-pixel image operators (similar to those from the actual example) are applied to an image. The choice of operators is static, and the image data is dynamic.

We expect that partial evaluation will improve the efficiency of these programs across all compiler types, since the program will be simplified beyond what we can expect from ordinary compiler optimizations.

11.5 Experiments

This section presents the experimental results; we first describe our benchmarking methodology, then give the result of benchmarking specialized programs produced as Java source, and last the result of benchmarking specialized programs produced as C source code. We only present the result of each benchmark; interpretation of the results is the subject of the next section.

11.5.1 Methodology

Experiments were performed on two different machines, a SPARC and an x86. The SPARC machine is a Sun Enterprise 250 running Solaris 2.7, with two 300MHz Ultra-SPARC processors (note that all benchmarks are single threaded). The x86 machine is a Dell OptiPlex GX1 running Linux 2.2, with a single 600MHz Pentium III processor.

All benchmarks are automatically specialized using JSpec. Compilation from Java source to Java bytecode is done using Sun's JDK 1.1 `javac` compiler with optimization selected (JSpec by default uses Sun's JDK 1.1 tools to produce a set of specialized class files). The JIT benchmarks are performed on SPARC using Sun's JDK 1.2.2 JIT compiler [149], and on x86 using IBM's JDK 1.3 JIT compiler [80]. The adaptive compilation benchmarks are performed using the HotSpot compiler included with Sun's JDK 1.3 beta 2 [151], available both for SPARC and x86. The interpreter benchmark is done using Sun's HotSpot compiler in interpretive mode. The off-line compilation benchmarks are done using the Harissa bytecode compiler [117], with optimization level `EO3`, except for the ChkPt benchmark where optimization level `EO1` was selected to limit resource consumption during compilation. The maximal heap size was set to 96Mb for all systems except Harissa (Harissa does not provide any means of limiting the amount of memory allocated by the program).

Each benchmark is structured as follows: a benchmark performs ten main iterations that are timed individually, and each such main iteration consists of some number of minor iterations of the actual test. The first five main iterations are discarded to allow JIT and adaptive compilers to optimize the program. All execution times are reported in seconds, and the number of minor iterations is adjusted to ensure that each main iteration runs long enough to give consistent time measurements. All benchmarks compute and print a checksum value which is threaded through the computation of each iteration of the benchmark, to prevent compiler optimizations such as loop invariant removal or dead code elimination from removing the code that is being benchmarked.

For the lack of automatic generation of guards by the specialization class compiler, the call to the specialization entry point is manually created. The insertion of guards in the program would have a minimal influence on the performance of the program, since guards in all cases can be inserted outside the critical regions of the program. All benchmarks except ChkPt are automatically specialized; due to implementation limitations, the ChkPt benchmark requires patching after specialization (for more details, see Chapter 12). For the ChkPt benchmark results are only reported with inlining optimization, since we use specialized Java code provided by the authors of this program, which is only available with inlining optimization.

11.5.2 Java-to-Java specialization results

We now present the results of using JSpec to specialize each of the programs listed in the previous section, generating AspectJ source code as output. We first consider the size of specialized programs, then the speedup due to partial evaluation on interpreters, JITs, adaptive compilers, and last off-line compilers. Afterwards, we compare the speedups due to partial evaluation first for the SPARC architecture and then for the x86 architecture. In all cases, experiments are done both with and without the post-specialization inlining optimization.

Program size

We measure the size increase in a Java program due to partial evaluation by the number of source code lines in the AspectJ code that defines the specialized aspect of the program. The number of lines needed in AspectJ syntax to express the introduction of a method in a class is roughly equivalent to the number of lines added to this class, so we consider this measure sufficiently precise for our purposes. The results are shown in Table 11.1; P-LOC$_G$ is the number of lines of code of the program, A-LOC$_S$ the number of lines of code for the specialized aspect, and I$_S$ is the size increase due to partial evaluation (i.e., I$_S = 2$ implies that program size is doubled); A-LOC$_{S+I}$ and I$_{S+I}$ are the corresponding numbers with the post-specialization inlining optimization activated. To show the effect of the inlining optimization on the size of the specialized program, the number I$_{-I}$ is the size increase due to *not* performing the inlining optimization.

In most cases, the size increase due to partial evaluation ranges from around two times to around six times. In the FFT and Fold:LS benchmarks, the size of the specialized program depends on the size of the input (for FFT, the size of the radix), and its size can thus be arbitrarily large. In all cases but one, the inlining optimization decreases the size of the program; most methods are used only once in the program, and inlining them decreases the overall size of the program but increases the size of each remaining method. The size decrease is primarily due to the elimination of numerous small, specialized methods, but also due to a simplification of the method call sites.

Interpreter speedup

The speedup due to partial evaluation with execution on an interpreter is shown in Table 11.2. Results are shown for SPARC and x86 architectures. The time to execute the generic program is denoted T_{Gen}, the time to execute the specialized program is denoted T_{Spec}, and the speedup factor due to partial evaluation ($T_{\text{Gen}}/T_{\text{Spec}}$) is denoted SF. Similarly, the time to execute the specialized program optimized with inlining is denoted $T_{\text{S+I}}$, and the speedup factor for this program is denoted SF$_{+I}$.

We observe that partial evaluation gives a significant speedup on both SPARC and x86 for the FFT, Bridge, ArithInt, Power-2 and Strategy experiments, and that the post-specialization inlining optimization greatly improves the performance of the specialized program. For many programs, partial evaluation results in a loss of performance; as explained in the next section, we expect a lack of post-specialization optimizations to be the cause of the negative results.

Name	P-LOC$_G$	A-LOC$_S$	$\mathbf{I}_S$	A-LOC$_{S+I}$	$\mathbf{I}_{S+I}$	I$_{-I}$
FFT:16	148	238	**2.61**	(1)	(1)	(1)
FFT:32	148	612	**5.14**	(1)	(1)	(1)
FFT:64	148	1508	**11.19**	(1)	(1)	(1)
Power-1	13	15	**2.15**	(1)	(1)	(1)
Bridge	235	314	**2.34**	186	**1.79**	1.31
Builder	282	265	**1.94**	252	**1.89**	1.03
Iterator	205	289	**2.41**	183	**1.89**	1.27
ArithInt	59	139	**3.36**	52	**1.88**	1.79
ChkPt	1114	(2)	(2)	457	**1.41**	(2)
Fold:LS	33	1324	**41.12**	1399	**43.39**	0.95
Fold:OP	33	78	**3.36**	(1)	(1)	(1)
IP:3	847	1282	**2.51**	784	**1.93**	1.30
IP:5	847	2914	**4.44**	1703	**3.01**	1.48
Pipe	97	115	**2.19**	47	**1.48**	1.48
Power-2	59	303	**6.14**	85	**2.44**	2.52
Strategy	238	215	**1.90**	130	**1.55**	1.23

(1): No significant opportunity for inlining optimization
(2): Experiment only performed with inlining optimization

Table 11.1: Specialization effect on code size measured in number of lines (selected program slice only).

	SPARC					x86				
Name	T_{Gen}	T_{Spec}	**SF**	$T_{\mathrm{S+I}}$	**SF**$_{+I}$	T_{Gen}	T_{Spec}	**SF**	$T_{\mathrm{S+I}}$	**SF**$_{+I}$
FFT:16	21.27	11.70	**1.82**	(1)	(1)	11.53	7.05	**1.64**	(1)	(1)
FFT:32	48.12	31.68	**1.52**	(1)	(1)	28.81	19.54	**1.47**	(1)	(1)
FFT:64	108.06	77.39	**1.40**	(1)	(1)	69.10	47.16	**1.47**	(1)	(1)
Power-1	48.27	19.42	**2.49**	(1)	(1)	7.94	3.36	**2.36**	(1)	(1)
Bridge	42.99	44.55	**0.96**	33.61	**1.28**	12.70	13.37	**0.95**	10.32	**1.23**
Builder	45.74	70.52	**0.65**	62.63	**0.73**	11.73	20.54	**0.57**	21.14	**0.55**
Iterator	52.86	77.85	**0.68**	53.50	**0.99**	15.56	25.96	**0.60**	18.41	**0.85**
ArithInt	229.16	214.97	**1.07**	115.24	**1.99**	59.43	58.33	**1.02**	36.18	**1.64**
ChkPt	25.74	(2)	(2)	26.82	**0.96**	7.86	(2)	(2)	8.93	**0.88**
Fold:LS	148.92	162.42	**0.92**	158.29	**0.94**	27.61	30.10	**0.92**	30.93	**0.89**
Fold:OP	167.97	135.90	**1.24**	(1)	(1)	33.58	24.29	**1.38**	(1)	(1)
IP:3	37.08	38.03	**0.97**	31.24	**1.19**	14.49	14.92	**0.97**	13.14	**1.10**
IP:5	96.54	96.82	**1.00**	79.44	**1.22**	34.23	37.92	**0.90**	31.07	**1.10**
Pipe	64.09	59.64	**1.07**	65.14	**0.98**	20.01	16.25	**1.23**	12.48	**1.60**
Power-2	438.63	336.38	**1.30**	131.20	**3.34**	90.78	68.89	**1.32**	19.84	**4.58**
Strategy	100.05	78.97	**1.27**	53.61	**1.87**	27.95	22.83	**1.22**	18.63	**1.50**

(1): No significant opportunity for inlining optimization
(2): Experiment only performed with inlining optimization

Table 11.2: Specialization benchmarks, interpreter execution using Sun's JDK 1.3 HotSpot compiler in interpretive mode.

	SPARC					x86				
Name	T_{Gen}	T_{Spec}	**SF**	$T_{\text{S+I}}$	**SF**$_{+I}$	T_{Gen}	T_{Spec}	**SF**	$T_{\text{S+I}}$	**SF**$_{+I}$
FFT:16	2.95	0.86	**3.42**	(1)	(1)	3.99	3.31	**1.20**	(1)	(1)
FFT:32	5.52	2.64	**2.09**	(1)	(1)	10.91	9.45	**1.16**	(1)	(1)
FFT:64	10.78	10.19	**1.06**	(1)	(1)	27.82	24.92	**1.12**	(1)	(1)
Power-1	5.26	4.28	**1.23**	(1)	(1)	1.00	0.91	**1.10**	(1)	(1)
Bridge	2.97	2.96	**1.00**	2.45	**1.21**	2.67	2.72	**0.98**	2.41	**1.11**
Builder	5.40	3.89	**1.39**	4.40	**1.23**	1.33	0.89	**1.49**	0.89	**1.49**
Iterator	4.46	4.08	**1.09**	3.59	**1.24**	1.34	0.95	**1.41**	0.72	**1.87**
ArithInt	24.18	7.06	**3.42**	5.46	**4.43**	8.05	0.91	**8.81**	0.35	**23.13**
ChkPt	6.39	(2)	(2)	4.68	**1.37**	1.93	(2)	(2)	1.94	**0.99**
Fold:LS	36.24	20.42	**1.77**	19.07	**1.90**	2.29	1.16	**1.97**	1.13	**2.03**
Fold:OP	48.95	42.81	**1.14**	(1)	(1)	6.23	4.45	**1.40**	(1)	(1)
IP:3	4.13	3.54	**1.17**	3.26	**1.27**	(3)	(3)	(3)	(3)	(3)
IP:5	9.38	10.51	**0.89**	9.06	**1.04**	(3)	(3)	(3)	(3)	(3)
Pipe	6.78	3.58	**1.89**	3.07	**2.21**	2.00	0.50	**3.99**	0.57	**3.51**
Power-2	38.13	5.82	**6.55**	5.01	**7.60**	7.39	1.38	**5.34**	1.42	**5.22**
Strategy	7.93	3.75	**2.11**	3.17	**2.50**	2.78	1.39	**2.00**	1.28	**2.17**

(1): No significant opportunity for inlining optimization
(2): Experiment only performed with inlining optimization
(3): Experiment failed due to a compiler bug

Table 11.3: Specialization benchmarks, JIT execution using Sun's JDK 1.2.2 and IBM's JDK 1.3 compilers.

JIT speedup

The speedup due to partial evaluation with execution on a JIT compiler is shown in Table 11.3; the table is labelled in the same way as for the interpreter benchmarks. For SPARC, partial evaluation improves performance for all benchmarks except IP:5; for x86, performance is improved for all benchmarks except ChkPt. (Note that IBM's JIT compiler failed to execute the IP experiments.) For both architectures, there are significant speedups for Builder, ArithInt, Fold:LS, Pipe, Power-2, and Strategy. Moreover, for SPARC, there are significant speedups for the FFT:16 and FFT:32 benchmarks. As well, for x86, there are significant speedups for the Iterator and Fold:OP benchmarks. For both architectures, the post-specialization inlining optimization gives significant speed improvements in most cases, but is detrimental to performance in a few cases.

Adaptive compiler speedup

The speedup due to partial evaluation with execution on an adaptive compiler is shown in Table 11.4; the table is labelled in the same way as for the interpreter benchmarks. For both architectures, partial evaluation improves performance almost everywhere. The only exception for both architectures is the IP:5 benchmark where performance is seriously degraded when the post-specialization inlining optimization is used; on SPARC partial evaluation does not improve the performance of the Power-1 benchmark, and on x86 it does not improve the performance of the Builder and Iterator benchmarks. For both architectures, there are significant speedups for the Builder, ArithInt, ChkPt, Fold:LS, Pipe, Power-2, and Strategy benchmarks. Moreover, for SPARC, there are significant speedups for the FFT, Iterator, and IP benchmarks; we consider

	SPARC					x86				
Name	T_{Gen}	T_{Spec}	**SF**	$T_{\text{S+I}}$	**SF**$_{+I}$	T_{Gen}	T_{Spec}	**SF**	$T_{\text{S+I}}$	**SF**$_{+I}$
FFT:16	1.55	0.25	**6.31**	(1)	(1)	4.61	4.07	**1.13**	(1)	(1)
FFT:32	2.71	0.62	**4.37**	(1)	(1)	12.18	11.40	**1.07**	(1)	(1)
FFT:64	4.63	1.53	**3.02**	(1)	(1)	30.58	30.02	**1.02**	(1)	(1)
Power-1	3.35	4.11	**0.81**	(1)	(1)	1.28	0.91	**1.41**	(1)	(1)
Bridge	4.07	4.01	**1.01**	3.95	**1.03**	1.18	0.88	**1.34**	0.96	**1.23**
Builder	2.36	1.30	**1.82**	1.32	**1.78**	1.50	0.64	**2.36**	1.50	**1.00**
Iterator	3.54	1.09	**3.24**	1.12	**3.16**	0.63	0.71	**0.88**	0.57	**1.11**
ArithInt	13.82	1.39	**9.94**	1.40	**9.86**	9.50	0.79	**12.07**	0.80	**11.92**
ChkPt	4.80	(2)	(2)	3.31	**1.45**	1.44	(2)	(2)	1.01	**1.42**
Fold:LS	32.35	0.68	**47.57**	0.90	**35.94**	4.82	1.31	**3.67**	1.03	**4.67**
Fold:OP	42.92	41.67	**1.03**	(1)	(1)	8.41	6.15	**1.37**	(1)	(1)
IP:3	2.87	1.33	**2.16**	1.41	**2.03**	2.87	2.53	**1.13**	2.54	**1.13**
IP:5	6.71	2.48	**2.71**	67.56	**0.10**	4.56	3.62	**1.26**	30.81	**0.15**
Pipe	5.80	1.75	**3.31**	1.76	**3.30**	2.20	0.53	**4.16**	0.53	**4.17**
Power-2	23.15	4.80	**4.83**	4.75	**4.87**	7.67	1.39	**5.51**	1.39	**5.51**
Strategy	5.58	1.65	**3.39**	1.63	**3.42**	3.36	1.64	**2.05**	1.64	**2.05**

(1): No significant opportunity for inlining optimization
(2): Experiment only performed with inlining optimization

Table 11.4: Specialization benchmarks, adaptive compiler execution using Sun's JDK 1.3 HotSpot compiler.

the extremely high speedup for the Fold:LS benchmark on SPARC to be valid but artificial, as explained in the next section. As well, for x86, there are significant speedups for the Bridge and Fold:OP benchmarks. Across both SPARC and x86 architectures, the post-specialization inlining optimization never improves performance; in some cases, such as the Builder and IP:5 experiments, it even strongly degrades the performance.

Off-line compiler speedup

The speedup due to partial evaluation for execution with an off-line compiler is shown in Table 11.5; the table is labelled in the same way as for the interpreter benchmarks. For both architectures, partial evaluation improves performance everywhere, except for Iterator on x86. For both architectures, there are significant speedups for the Builder, ArithInt, ChkPt, Fold:LS, Pipe, Power-2, and Strategy benchmarks. Moreover, for SPARC, there are significant speedups for the FFT and IP benchmarks. As well, for x86, there are significant speedups for the Power-1 and Fold:OP benchmarks. With a single exception, the post-specialization inlining optimization never improves performance for both SPARC and x86 architectures; the only significant exception is ArithInt, where the inlining optimization in the compiler (Harissa) does not compensate for a lack of inlining optimization in the specialized program.

SPARC architecture speedup comparison

The speedup due to partial evaluation on SPARC architecture is compared for all execution environments in Figure 11.1; benchmarks where post-specialization inlining optimization has been performed appear twice in the figure, with "+I" indicating the benchmark where inlining was performed. In almost all cases,

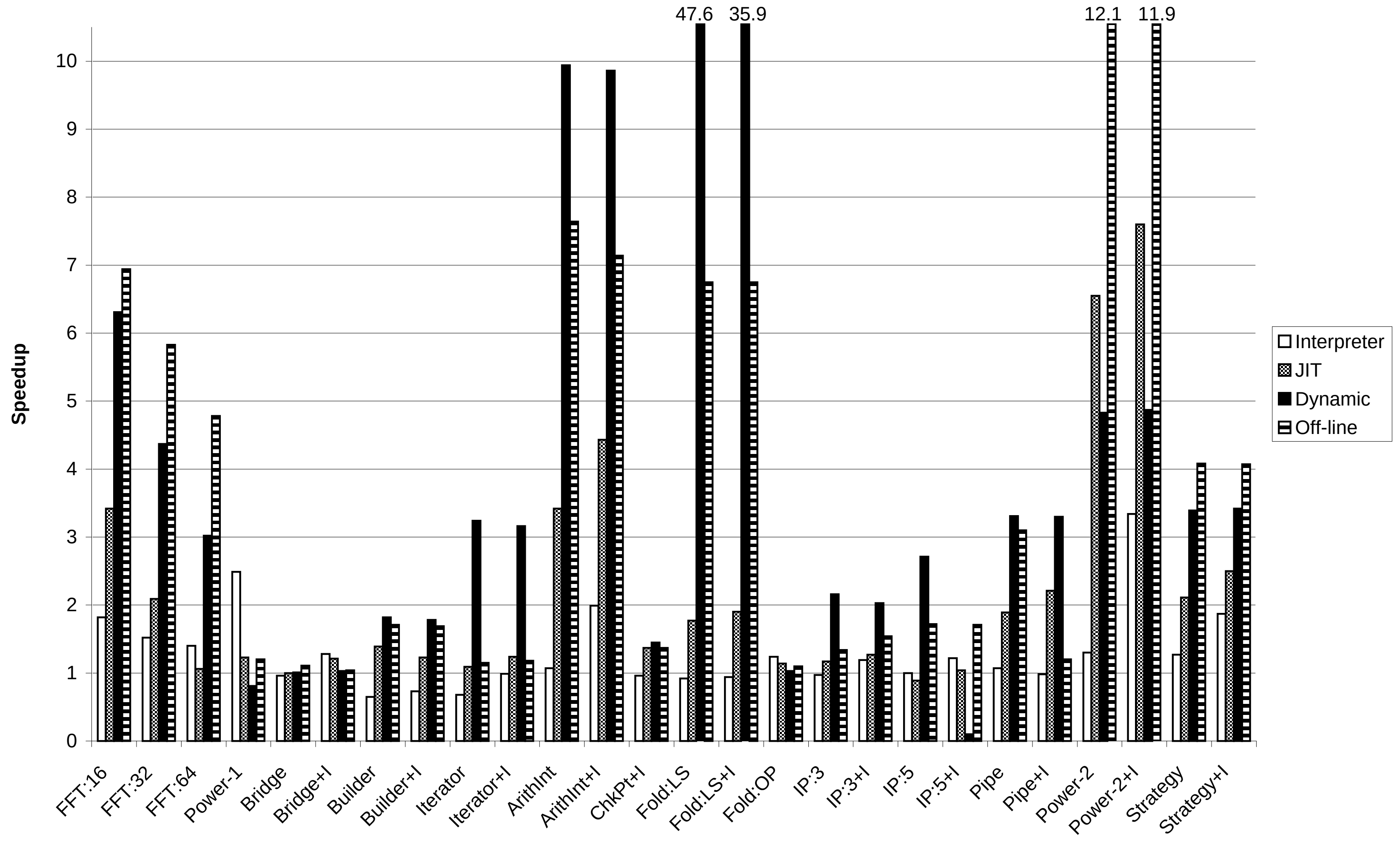


Figure 11.1: Speedup comparison, SPARC ("+I" indicates benchmark with inlining optimization).

	SPARC					x86				
Name	T_{Gen}	T_{Spec}	**SF**	$T_{\text{S+I}}$	$\mathbf{SF}_{+I}$	T_{Gen}	T_{Spec}	**SF**	$T_{\text{S+I}}$	$\mathbf{SF}_{+I}$
FFT:16	1.34	0.19	**6.94**	(1)	(1)	4.11	3.29	**1.25**	(1)	(1)
FFT:32	2.27	0.39	**5.83**	(1)	(1)	11.16	9.46	**1.18**	(1)	(1)
FFT:64	4.12	0.86	**4.78**	(1)	(1)	28.48	25.01	**1.14**	(1)	(1)
Power-1	4.16	3.47	**1.20**	(1)	(1)	1.14	0.67	**1.70**	(1)	(1)
Bridge	38.52	34.78	**1.11**	37.04	**1.04**	25.24	24.58	**1.03**	24.64	**1.02**
Builder	4.36	2.55	**1.71**	2.58	**1.69**	1.50	1.19	**1.26**	1.18	**1.27**
Iterator	4.80	4.16	**1.15**	4.06	**1.18**	1.51	1.34	**1.12**	1.91	**0.79**
ArithInt	22.91	3.00	**7.64**	3.21	**7.14**	8.93	1.67	**5.34**	1.07	**8.38**
ChkPt	2.07	(2)	(2)	1.52	**1.37**	1.39	(2)	(2)	0.95	**1.45**
Fold:LS	32.70	4.85	**6.75**	4.85	**6.75**	5.21	1.70	**3.07**	1.71	**3.04**
Fold:OP	45.49	41.28	**1.10**	(1)	(1)	12.01	6.99	**1.72**	(1)	(1)
IP:3	2.38	1.78	**1.34**	1.54	**1.54**	2.04	1.82	**1.12**	1.80	**1.13**
IP:5	4.21	2.45	**1.72**	2.46	**1.71**	3.93	3.31	**1.19**	3.28	**1.20**
Pipe	5.32	1.71	**3.10**	4.41	**1.21**	2.10	0.69	**3.06**	0.74	**2.82**
Power-2	37.92	3.25	**11.68**	3.28	**11.56**	9.72	1.54	**6.30**	1.28	**7.58**
Strategy	5.04	1.24	**4.08**	1.24	**4.07**	3.51	2.07	**1.70**	2.03	**1.73**

(1): No significant opportunity for inlining optimization
(2): Experiment only performed with inlining optimization

Table 11.5: Specialization benchmarks, off-line compiler execution using Harissa.

	SPARC					x86				
Name	T_{Gen}	T_{SpecJ}	**SF**	T_{DSpec}	$\mathbf{SF}_{DSpec}$	T_{Gen}	T_{SpecJ}	**SF**	T_{DSpec}	$\mathbf{SF}_{DSpec}$
FFT:16	1.34	0.19	**6.94**	0.16	**8.16**	4.11	3.29	**1.25**	3.40	**1.21**
Builder	4.36	2.58	**1.69**	1.93	**2.26**	1.50	1.18	**1.27**	0.77	**1.95**
IP:5	4.21	2.45	**1.72**	1.96	**2.15**	3.93	3.31	**1.19**	3.11	**1.27**

Table 11.6: Specialization with C source code as output (execution under Harissa).

the benefit due to partial evaluation is largest for the adaptive and off-line compilers. For these compilers, there are many programs with significant speedups. In terms of absolute execution speed, the adaptive compiler and the off-line compiler produce the fastest code. (The absolute execution speed is not shown in the figure, but can be seen in Tables 11.2, 11.3, 11.4, and 11.5.)

x86 architecture speedup comparison

The speedup due to partial evaluation on x86 architecture is compared for all execution environments in Figure 11.2; the "+I" notation is used again to indicate benchmarks with post-specialization inlining optimization. The benefits due to partial evaluation vary in the same way for the JIT, adaptive and off-line compilers. In terms of absolute execution speed, the interpreter is slowest, the adaptive compiler and the off-line compiler give similar performance, and by a slight margin the JIT compiler produces the fastest code. (Again, the absolute execution speed is not shown in the figure, but can be seen in Tables 11.2, 11.3, 11.4, and 11.5.)

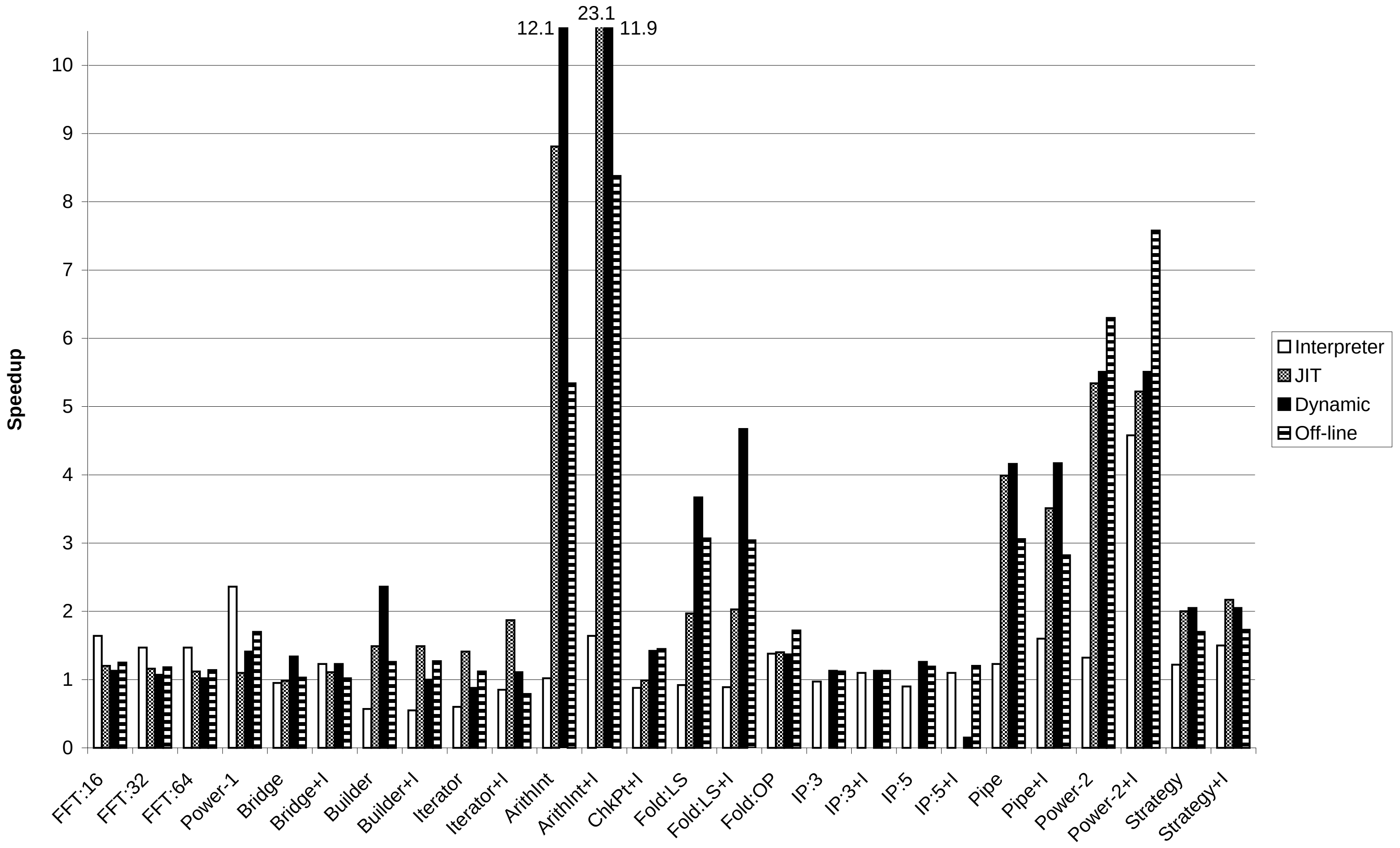


Figure 11.2: Speedup comparison, x86 ("+I" indicates benchmark with inlining optimization).

11.5.3 Java-to-C specialization results

To evaluate the advantage of generating the specialized program as C source code and executing it directly under the Harissa environment, we compare the performance of a set of specialized programs generated both as Java source code and as C source code. We use the FFT:16, Builder, and IP:5 benchmarks; in these benchmarks a significant number of array accesses and downwards type casts are residualized, which makes them an interesting target for this kind of specialization. We compare the best running time (inlining/no inlining) measured for executing the specialized Java source code compiled using Harissa (denoted T_{SpecJ}) to the running time for the specialized C source code executing directly under Harissa (T_{DSpec}). The speedup due to specialization with direct execution under the Harissa environment (relative to the generic program) is referred to as SF_{DSpec}. The results are shown in Table 11.6.

For both architectures, direct execution of specialized C code under the Harissa environment provides a significant advantage for the Builder benchmark. Moreover, for SPARC, there are significant advantages for the FFT:16 and IP:5 benchmarks.

11.6 Assessment

The benchmarks presented in the previous section show that the benefit due to partial evaluation varies with the specific experiment, the choice of Java execution model, and the machine architecture. In this section, we analyze the benchmark results and present our conclusions. We first discuss the impact of the kind of specialization opportunity found in the program, and then discuss the impact of the execution model and the machine architecture.

11.6.1 Impact of the kind of specialization opportunity

We have classified the benchmark programs by the kind of specialization opportunity that we exploit using partial evaluation, namely imperative, object-oriented, and mixed. We now discuss the overall effect of partial evaluation on each of these kinds of programs.

Imperative specialization opportunities

The FFT benchmark significantly benefits from partial evaluation on the SPARC architecture, and hardly benefits from partial evaluation on the x86 architecture. The greatest speedup was obtained with off-line compilation on a SPARC; the speedup here is about equivalent to the speedup that has been observed for partial evaluation of FFT routines written in C [96]. The Power-1 benchmark is often used to illustrate partial evaluation for functional and imperative languages, and always with a gain in performance; we see that for Java there is usually a small gain, but that specialization can be detrimental as was the case for SPARC with adaptive compiler execution.

Object-oriented specialization opportunities

Except for execution by interpretation, the Builder benchmark benefits from partial evaluation across all execution models and architectures. As well, the Iterator benchmark benefits from partial evaluation across most execution models and architectures. On the contrary, the Bridge benchmark only benefits from partial evaluation on an interpreter. There is a more widespread use of virtual dispatches to enhance program adaptation in the Builder and Iterator benchmarks than in the Bridge benchmark, which we believe makes them more difficult to optimize.

Mixed specialization opportunities

Disregarding execution by interpretation, partial evaluation is beneficial for most of the programs with mixed specialization opportunities, as was expected. The ArithInt, Fold:LS and Power-2 benchmark programs are greatly simplified by partial evaluation, resulting in a large speedup. We believe the extremely high speedup observed for the Fold:LS benchmark on SPARC to be due to arithmetic simplifications performed by the compiler; it is however intrinsic to the program that it can be optimized in this way, and so it is a valid benchmark. Also, our results for the ChkPt benchmark correspond with the results reported by the authors of this program [97].

The IP benchmark programs are also greatly simplified by partial evaluation, but the speedup is in most cases small or nonexistent. This observation does not correspond to the results reported in earlier work, where experiments were done with the same application [143]. In this paper, partial evaluation was reported to yield a 1.5 to 5 times speedup. These results were obtained by directly generating C code and by manually back-translating the C code into Java. Rerunning the same code using the Java compilers of this chapter gives a smaller speedup, although the results are better than what we have obtained in this chapter. We attribute this difference to a number of factors. First, the Java compilers have evolved, and are better at handling generic object-oriented code; in addition, these compilers may have been further tuned to better handle typical Java programs, which are different from specialized Java programs. Second, the manual translation from C to Java inadvertently optimized the program in some cases using common-subexpression elimination; improving the back-translator to include the same optimization should improve the execution speed of the IP benchmark. Last, the specialized program used in the earlier work was optimized using the Tempo SUIF-based post-processing [41, 166], but this back-end fails on the intermediate specialized C code currently generated by JSpec; using more back-end optimizations with JSpec should improve the execution speed of the IP benchmark. We consider these issues to be future work.

11.6.2 Impact of execution model and architecture

With a few notable exceptions, the specialization of Java programs for execution on an interpreter is detrimental to performance. Given that optimization of Java programs for execution on embedded systems represents an interesting domain of applications for partial evaluation (as described in Chapter 13), we find these results disappointing. We believe the poor results to be due to a lack of post-specialization optimizations. Thus far we have concentrated on replacing expensive instructions by cheaper ones (e.g., replacing a virtual dispatch by a direct dispatch). However, on an interpreter, the total number of instructions executed can be more significant than their individual cost. Aggressive optimizations such as object inlining combined with common subexpression elimination and copy propagation would probably improve performance. However, we consider it future work to determine what analyses and transformations are needed to improve execution speed on Java interpreters.

Specialization with direct output as specialized C source code provided a definite advantage on the selected subset of benchmarks in Section 11.5.3. This observation suggests that integration of a partial evaluator with a specific compiler (either off-line or adaptive) can be a definite advantage in terms of performance of the specialized program. However, as is the case for Harissa, the platform-independence advantage of Java is lost when generating C code for use with a specific compiler implementation.

On the SPARC architecture, adaptive compilation and off-line compilation benefit more from partial evaluation than JIT compilation, which again benefits more from partial evaluation than interpretation. Here, it seems that the more aggressive the compiler, the greater the benefit from partial evaluation. On the x86 architecture, the picture is slightly different. The JIT compiler generates the fastest-running code (both for generic and specialized programs), and is likely to be as aggressive in its optimization strategy as the adaptive and off-line compiler. (There is no less aggressive compiler suitable for comparison, since both the adaptive compiler and the off-line compiler perform similar optimizations.) We conclude that programs specialized with partial evaluation benefit from aggressive compilation, and conjecture that to some extent, the more aggressive the compiler the higher the benefit from partial evaluation. However, to support this claim a greater selection of Java compilers should be tested; we consider such a study to be future work.

This experimental study has also revealed another important issue: specialized programs generated by partial evaluation stress different parts of a compiler and its run-time system than standard programs. Initial benchmarks revealed that the speedup due to partial evaluation when executing Harissa-compiled code was negligible in many cases. The problem was traced to a highly inefficient implementation of Java type-cast checking. Specialized code often contains many type casts (a type cast is used to express a direct method invocation), whereas standard Java code rarely contains type casts. Optimization of the Harissa type-cast checking implementation resulted in the performance numbers reported in the previous section, where partial evaluation gives a significant speedup in most cases.

Comparing across architectures, we see that the speedup due to partial evaluation is greater on SPARC than on x86. SPARC is a RISC architecture; as such, it relies on the compiler to generate code that executes efficiently. In contrast, the x86 architecture traditionally has more built-in logic to help dynamically optimize the program execution [153]. As a consequence, the code simplifications brought about by partial evaluation have a larger impact on SPARC than on x86, because partial evaluation simplifies the program based on knowledge about the run-time behavior of the program, and thereby helps the compiler generate code already adapted to the run-time behavior of the program.

These compiler and architecture issues suggest that the choice of Java run-time environment and machine architecture must be taken into account when deciding how to specialize a program. This need is contrary to the platform-independent nature of Java and complicates the use of partial evaluation. The inlining optimization can be omitted in most cases, but there is still a choice of what parts of the input data of the program to consider static, and of whether to specialize at all. When information about where a program will be run is not available, partial evaluation should probably be used in a conservative way when applied to Java programs. To further guide the user in how partial evaluation can be safely used, experience with the speedup due to partial evaluation on different architectures and execution models can be integrated into the specialization pattern approach. In addition, if the partial evaluator supports run-time specialization of Java bytecode (see Chapter 13), partial evaluation can dynamically adapt to the compilation system that it is running on, perhaps through the use of specialization class declarations to describe different scenarios. We consider the development of such techniques to be future work.

11.7 Summary

Faced with the large number of unknown factors in the performance of specialized Java programs, we have in this chapter presented an experimental study of the effect of partial evaluation on Java programs. We address the impact of the choice of Java execution model and machine architecture on the performance of the specialized program, and the different kinds of specialization opportunities that exist in Java programs.

We conclude that partial evaluation as implemented by JSpec can significantly optimize the performance of programs. For Java, partial evaluation is beneficial across most programs when using adaptive and off-line compilers, mostly beneficial when using JIT compilers, and usually a disadvantage when using an interpreter. Furthermore, the benefit due to specialization is on the average observed to be larger on the SPARC architecture than on the x86 architecture. We also observe that in many cases the benefit due to the inlining optimization is nonexistent or negligible. However, more experimental and analytical work is needed to provide a clearer assessment of the speedup that results from partial evaluation.

Part IV

Perspectives

Chapter 12

Related Work

12.1 Introduction

The natural ease with which generic programs are implemented in object-oriented languages combined with the overhead that results from heavy use of object-oriented features has made object-oriented languages a natural target for research in specialization techniques. Several completely automated specialization techniques have been developed in previous work (some of these were presented in Chapter 3), and various forms of partial evaluation have been investigated for object-oriented languages. Nonetheless, these approaches do not give a complete picture of how partial evaluation fits into the object-oriented paradigm.

This chapter presents related work not already discussed in this document, and elaborates on the relation between object-oriented partial evaluation and selective argument specialization. The existing work in partial evaluation for object-oriented languages is limited, but is complemented by existing work in specialization techniques for optimizing compilers.

Chapter overview: First, Section 12.2 presents alternative approaches to defining partial evaluation for object-oriented languages. Section 12.3 discusses related work in partial evaluation techniques. Then, Section 12.4 investigates the exact relationship between partial evaluation and certain compiler optimizations. Section 12.5 compares partial evaluation to templates. Afterward, Section 12.6 describes work that is related to specialization patterns. Last, Section 12.7 presents a summary.

12.2 Object-Oriented Partial Evaluation

Partial evaluation has been investigated for object-oriented languages both through the development of techniques for partial evaluation and through experiments in applying partial evaluators within other domains. We first present related work in partial evaluation for object-oriented languages, and then discuss an application of object-oriented partial evaluation to optimize fault-tolerant computing.

12.2.1 Partial evaluation

Related work in the development of partial evaluation for object-oriented languages includes run-time specialization for C++ programs based on internal object state, on-line partial evaluation for the object-based language Emerald, an object-oriented equivalent of type-directed partial evaluation named interface-directed partial evaluation, and a partial evaluator for Java bytecode without objects.

Run-time specialization of C++

Partial evaluation can be done at run time based on constructor parameters for C++ programs, as shown by Fujinami [61, 62]. Here, annotations are used to indicate member methods that are to be run-time specialized, and specialization is done for all class members that are constant or protected by encapsulation. A method is specialized by simplifying imperative computations that rely on known values using standard partial evaluation principles, and by replacing virtual dispatches through static object references by direct method invocations. Furthermore, if a virtual method invoked through a static object reference has been tagged as `inline`, it is inlined into the caller method and further specialized. Guards are automatically generated; they aim to respecialize methods when specialization invariants are invalidated.

This approach to partial evaluation for an object-oriented language is akin to instantiation-time specialization, and provides a fine degree of control on what program slice is included in the specialization. However, only a single object (and any code that can be inlined into it) is specialized, and only the state of this object is taken into account. On the contrary, we specialize the interaction that takes place between multiple objects based on their respective state, even when performing instantiation-time specialization. Compared to Fujinami's partial evaluator, JSpec can be harder to control, but a wider range of specialization scenarios can be exploited since JSpec incorporates a much richer set of analyses and transformations.

Partial evaluation for Emerald

On-line partial evaluation for object-oriented languages is feasible, as shown by Marquard and Steensgaard [107]. They developed a partial evaluator for a small object-based object-oriented language based on Emerald. The primary focus is on issues in on-line partial evaluation, such as termination and resource consumption during specialization. There are no considerations on how partial evaluation should specialize an object-oriented program, and there is virtually no description of how their partial evaluator handles object-oriented language features. Nevertheless, it appears that they analyze and specialize programs according to the principles for object-based languages outlined in Chapter 8. In contrast, we concentrate on realistic class-based languages, and our goal is to take all aspects of the object-oriented paradigm into account in our definition of partial evaluation for object-oriented languages.

Interface-directed partial evaluation for Java

Inspired by type-directed partial evaluation for functional languages [45], interface-directed partial evaluation (IDPE) for Java has been defined by Benaissa and Tolmach [13]. IDPE specializes a small subset of Java without ordinary control structures, side effects and base-type values; specialization is done similar to type-directed partial evaluation for functional languages. Although IDPE serves to specialize programs written in the targeted Java language subset, little attention is paid to the object-oriented features of Java. In particular, virtual dispatches are not addressed, and there is no obvious way of using IDPE to specialize programs written in an object-oriented style of programming.

Partial evaluation for Java bytecode

In an unpublished note, Bertelsen describes an off-line partial evaluator for a Java bytecode subset without objects [16]. A constraint-based binding-time analysis is presented, and a design for a specializer is described. However, the essential issue of partial evaluation of a language incorporating object-oriented mechanisms is not addressed.

12.2.2 Application: fault-tolerant computing

The JSpec partial evaluator has been applied by Lawall and Muller to optimize checkpointing routines that save the current state of an object structure [97]. Here, it is explained how to implement a general checkpointing routine that can be specialized for information regarding what parts of the object structure are invariant in a specific execution stage of the program. Specialization classes are used to declare invariant structure parts; after specialization, the checkpointing routine no longer traverses this data. To measure the effect of specialization on an application, a checkpointing routine for a binding-time analysis is specialized. This experiment is the `ChkPt` benchmark reported in Chapter 11.

Due to the monovariance of the alias analysis, JSpec cannot currently specialize this benchmark program without minor manual modifications to enhance the result of the binding-time analysis. Objects of static and dynamic binding times are accessed using common accessor functions, and a polyvariant alias analysis is needed to precisely determine the patterns of object access in the program. We consider it future work to enhance the JSpec binding-time analysis so that standard checkpointing routines can be specialized.

12.3 Partial Evaluation Principles

There are a number of partial evaluation principles not directly targeted to specialization of complete object-oriented programs but that nonetheless are relevant to the material presented in this document. Specifically, specialization via translation has been investigated before, partial evaluation of Java reflection mechanisms has been studied, and partial evaluation for functional languages can optimize method dispatching.

12.3.1 Specialization via translation

The JSpec partial evaluator is implemented using a translation-based approach. The approach of using translation to extend the applicability of an existing partial evaluator to new program constructs has been investigated by Moura [115]. She developed an approach to specializing imperative programs by translation into a functional subset of Scheme, followed by partial evaluation using the Schism partial evaluator [40]. The translation from C to Scheme is much more complicated than our translation from Java to C, and it seems unlikely that this approach would scale to partial evaluation of realistic programs written in C.

12.3.2 Specialization of reflection

Partial evaluation can specialize uses of Java reflection, as shown by Braux and Noyé [25]. They extend a standard (informally described) binding-time analysis to be able to specialize Java reflection. Using this approach, static uses of reflection are removed by specialization, and static-and-dynamic uses of reflection are reflected into their Java syntax counterparts. To enable reflection over dynamic data, type inferencing during binding-time analysis allows separate binding times to be computed for the type of an object and the value of an object. As a benchmark, a program using the Java serialization API is specialized; this specialization yields a speedup ranging from 1.2 to 1.6 times.

The JSpec binding-time analysis cannot derive separate binding times for the type and the value of an object. Here, the most interesting case is static type and dynamic value, which allows reflection operations over dynamic data to be specialized. However, an effect similar to a static type and a dynamic value can be simulated by using a static-and-dynamic object with dynamic fields, although extensive experiments would be needed to decide whether this approach would provide sufficiently precise binding-time information to specialize programs that use reflection. In any case, the reflection specialization actions can be integrated into the JSpec specialization phase, either by direct implementation or by a translation-based approach.

12.3.3 Optimizing method dispatching

Partial evaluation can be used as a means of eliminating virtual dispatches similar to customization [33], as shown by Khoo and Sundaresh [88]. Given the class of the self object, they use partial evaluation to replaces virtual dispatches over the self with direct dispatches. JSpec performs similar simplification of virtual dispatches, but over all known object references, not only the self object.

12.4 Optimizing Compilers

Keeping in mind the discussion of the previous chapter on the fundamental differences between partial evaluators and compilers, we now compare compiler optimization techniques to partial evaluation. We first discuss customization and selective argument specialization, then rewrite-system based compilation, and last the direct use of partial evaluation techniques in compilers.

12.4.1 Selective argument specialization

Selective argument specialization was presented in Chapter 3, along with customization. Customization and selective argument specialization are general-purpose optimizations that specialize programs for local object type information; partial evaluation for object-oriented languages specifically targets generic programs, completely specializes a program for the concrete data that it manipulates, and requires user control. Nonetheless, it is interesting to try to compare the transformations that these techniques perform based on object type information.

Customization specializes a method to the set of its possible receivers; a method variant is generated for each possible receiver type, and each variant is specialized by eliminating virtual dispatches where possible based on information about the type of the self object [33]. Selective argument specialization generalizes customization to specialize for any method arguments (including the self object), and permits more aggressive optimization since specialization is concentrated on critical program parts [48]. Selective argument specialization relies on multi-dispatching in the run-time system to specialize methods for arguments other than the self object, and is directed by automatically gathered profiling information to avoid code explosion [49]. On the contrary, partial evaluation unconditionally specializes all virtual dispatches: when the self object is static, the virtual dispatch is removed, and the body of the receiver is specialized for information about the self object and any other arguments; when the self is static-and-dynamic, the virtual dispatch is made trivial to optimize, and the receiver is similarly specialized; when the self object is dynamic, a virtual dispatch is residualized, and each potential receiver is specialized for information about arguments other than the self object.

The key difference between partial evaluation and selective argument specialization is that partial evaluation only specializes for known, fixed data, whereas selective argument specialization speculatively specializes based on information either contained in the program or gathered using profiling techniques. As an example, consider how selective argument specialization and partial evaluation specialize for the self object argument. When the exact type of a receiver object is known, both selective argument specialization and partial evaluation remove virtual dispatches over the self. However, when there are a set of possible types, selective argument specialization specializes each virtual dispatch by specializing the potential receiver methods for each of the possible types of the self object, whereas partial evaluation considers the self object to be dynamic and does not specialize the potential receiver methods for information about the self object. As for specialization for arguments other than the self object, partial evaluation only specializes for these arguments when they are known. Nevertheless, we believe that partial evaluation could be generalized to work with multi-dispatching similarly to how selective argument specialization generalizes customization; we return to this point in the next chapter.

To some extent, partial evaluation and selective argument specialization are complementary. Partial evaluation could be extended to include an optimization similar to selective argument specialization that would improve specialization of receiver methods whenever the self object is dynamic. However, selective argument specialization already does a good job as a compiler optimization, and we would prefer to avoid increasing the complexity of the partial evaluator by adding compiler features. Unlike partial evaluation, selective argument specialization is a sufficiently restrained specialization technique that it can be performed without user intervention. These considerations help in defining the role of partial evaluation for object-oriented languages; there already exist fully automatic general-purpose specialization systems, and partial evaluation should not aim to duplicate their functionality. Rather, partial evaluation should be considered part of the software development process, as we argue in Part II of this document.

12.4.2 Rewrite-system based compilation

Architectural program transformations can be used before compilation to optimize uses of design patterns, as shown by Turwé and De Meuter [156]. Here, program rewriting techniques are used in a program transformation engine based on Prolog that optimizes uses of design patterns. While this optimization technique is very different from partial evaluation, with the appropriate rewriting rules, it can obtain local structural simplifications similar to those performed by partial evaluation. Since this technique does not rely on binding-time information, it can also be used to perform transformations beyond the capabilities of partial evaluation. However, as with all compiler techniques, the transformations performed by the compiler are limited by the precision of its analyses and the time that can be spent optimizing the program. In addition, new rewriting rules must be defined when new program patterns are invented. In the specific case of design patterns, their approach can be unified with specialization patterns: for each design pattern, a specialization pattern can describe what rewriting rules give the best optimizations.

12.4.3 Partial evaluation in a compiler

Partial evaluation techniques can be used to optimize scientific computation programs written in Java, as shown by Veldhuizen [160]. Here, a compiler that incorporates standard optimizations combined with partial evaluation techniques is used to eliminate virtual dispatches and perform object inlining. The result is a greatly simplified, compiled program; the optimizations performed by the compiler are similar to specialization based on expression templates in C++ (see next section). Nonetheless, as is the case for all compilers, the amount of optimization performed by the compiler is limited by the precision of its analyses and is hard to predict; overheads due to heavy use of object-oriented features are not guaranteed to be optimized away.

12.5 C++ Templates

The C++ language incorporates templates that allow a single class to be parameterized by types and values, and then be instantiated by the compiler to specific types and values. The instantiated classes can be seen as specialized versions of the generic class, although the generic class only serves as a template, and cannot be used in the program. In effect, templates in C++ allow the programmer to express static information about types and simple values, thus providing more information to the compiler. For example, rather than implementing the strategy pattern with a virtual call, the choice of strategy can be statically fixed using templates [64].

Combined with other C++ language features, templates can be used to perform partial evaluation at compile time, as demonstrated by Veldhuizen [159]. By using a combination of template parameters and C++ `const` constant declarations (which are guaranteed to be evaluated at compile time), arbitrary computations over base type values can be performed at compile time. In addition, entire parse trees of expressions can be encoded into template types, and then unfolded by the compiler to produce an efficient program [158]; this technique is referred to as *expression templates.*

Compared to our definition of partial evaluation for object-oriented languages, specialization with C++ templates is limited in a number of ways. First, the values that can be manipulated are more restricted; for example, objects cannot be dynamically allocated. Second, the computations that can be simplified are more limited; for example, virtual dispatches cannot be simplified. Last, there is no interaction with external program parts. In addition, since all computations are performed symbolically by a transformation engine not intended to be used as a partial evaluator, compilation of expression template programs can be a lengthy process. Furthermore, an explicit two-level syntax must be used to write programs; as a consequence, binding-time analysis must be performed manually, and functionality must be implemented twice if both a generic and a specialized behavior is needed.

Even if templates are limited compared to a standard partial evaluator, they have a significant advantage, namely predictability: it is guaranteed by the language specification that any computations at the template level are performed during compilation. Existing partial evaluators for two-level functional [152] and imperative [58] languages rely on similar annotations to ensure complete predictability. Nonetheless, we consider templates to be a source of inspiration for how object-oriented partial evaluation can be made more predictable.

12.6 Specialization Patterns

Specialization patterns serve both to eliminate systematically recurring overheads and to aid in the communication of knowledge about partial evaluation. In related work, partial evaluation has been used to eliminate recurring overheads introduced by the use of software architectures, and techniques for partial evaluation of interpreters have been documented.

12.6.1 Software architectures

A design pattern can be said to describe a micro-architecture that is implemented specifically for the program being developed. On the contrary, a software architecture defines a program-wide recurring code organization [145]. Marlet et al. have shown that program specialization can automatically eliminate the flexibility overhead of software architectures and generate an efficient implementation of the program infrastructure [106]. Due to constraints imposed on how a software architecture is used, specialization of a collection of programs written according to a given software architecture is often simpler than specialization of a collection of programs written using the same design patterns. All programs based on the same software architecture implementation can be specialized using the same specialization strategy, whereas specialization of a program written using a given design pattern depends on how the design pattern is used.

12.6.2 Specialization recipes

With the goal of communicating basic knowledge about partial evaluation, Jones describes a number of general techniques for specializing interpreters [83]. Since the effect of partial evaluation on a program can be seen as the elimination of a layer of interpretation, these techniques apply to partial evaluation of most kinds of programs. The generality of the techniques seems to suggest that specialization patterns should be complemented by a general description of how to apply partial evaluation, just like the presentation of design patterns by Gamma et al. [64] is complemented by general ideas on object-oriented design.

An in-depth investigation of how interpreters written in C can be specialized both at compile time and at run time is presented by Thibault et al. [154]. Most of the techniques can easily be adapted to object-oriented languages, and would make a solid basis for an interpreter specialization pattern.

12.7 Summary

Existing work in partial evaluation for Java has investigated specialization of object-oriented mechanisms, but has not concisely stated the complete effect of partial evaluation on a program. Related work in optimizing compilers is more extensive; specialization-like techniques are often used by optimizing compilers for object-oriented languages. Although these techniques are more limited, they offer certain advantages, most notably the independence of user guidance. However, partial evaluation is to a large extent complementary to optimizing compilers, which makes it interesting as a software engineering tool. As an interesting mix of these two approaches, C++ templates allows a standard C++ compiler to perform a restricted form of partial of partial evaluation.

Chapter 13

Future Work

13.1 Introduction

In this document we have investigated partial evaluation within the object-oriented paradigm of programming. Given the rich and diverse nature of object-oriented languages, we believe that object-oriented languages represent a new and interesting field of research for partial evaluation principles and techniques.

The purpose of this chapter is to present the immediate directions for future work opened by our investigation of object-oriented partial evaluation. We have deferred the exploration of many topics to future work; we now present those topics for which we have concrete ideas on how they can be pursued.

Chapter overview: First, Section 13.2 presents new concepts in object-oriented partial evaluation. Then, Section 13.3 presents future implementation work on JSpec. Section 13.4 presents future work in using partial evaluation as a software engineering technique. Last, Section 13.5 summarizes the contents of this chapter.

13.2 Concepts in Object-Oriented Partial Evaluation

So far, we have given a formal definition of partial evaluation for object-oriented languages. However, we have not yet developed any proof of correctness of our formalization. Furthermore, there are many language and partial evaluation features yet to be explored. In addition, based on our experiments with object-oriented partial evaluation, we are interested in partial evaluation as a tool for reducing the size of programs.

13.2.1 Correctness issues

We are interested in completing the formalization of object-oriented partial evaluation presented in Chapter 8 by proving that the binding-time analysis derives well-annotated programs, that well-annotated programs specialize into residual programs, and that specialization of a well-typed program again produces a well-typed program. Furthermore, with inspiration from existing proofs of the formal correctness of partial evaluation for first-order functional languages [42, 84], we are also interested in proving that object-oriented partial evaluation performs semantically correct specialization.

13.2.2 Language features

We have focused on class-based object-oriented languages throughout most of this document. However, as described in Chapter 8, we are interested in defining partial evaluation for object-based languages and in implementing a complete partial evaluator for such a language. The σ-calculus [1] may prove interesting as an initial target for partial evaluation, but partial evaluation should also be investigated for realistic languages such as Cecil [30, 32] or Self [157]. Cecil also includes multi-dispatching, which is likely to offer new kinds of opportunities for specialization; generic program functionality that in a language like Java is expressed using explicit type tests can be expressed in Cecil using multi-dispatching. Specialization of multi-dispatching is likely to be similar to specialization of standard virtual dispatching, but more work is needed to determine if similar partial evaluation analyses and transformations can be used.

We have not touched upon the issue of block structure in the language. Block structure and lexical scope were first introduced with Algol 60 [118], and were carried into the object-oriented paradigm by the first object-oriented language, Simula [121]. Block structure allows functionality local to a class to be declared locally, and when appropriate used elsewhere through a globally known interface. Block structure is fundamental to programming in Beta [102] and Simula, and is used extensively in Java [68]. However, partial evaluation is, in some cases, impeded by the presence of block structure in a program, and incorporating support for block structure into object-oriented partial evaluation has turned out to be non-trivial. In partial evaluation for functional languages, pre- and post-processing transformations named lambda-lifting [82] and lambda-dropping [47] have been used to support block structure [22, 40], and we are interested in developing similar transformation for object-oriented languages.

13.2.3 Partial evaluation features

The most challenging issue in using JSpec to specialize realistic programs is the difficulty in obtaining binding-time annotations that ensure the elimination of critical overheads in the program. Inspired by ongoing work in the Compose group [110], we would like to develop an approach that gives the user more control over the binding-time inference process by allowing binding times to be specified at critical program points. Specialization class declarations could be used to provide binding-time information, and the failure to produce binding times that match the user specification should be considered an error during specialization.

As was described in Chapter 11, the high number of type casts in residual programs produced by JSpec can be a serious overhead in the execution of the program. In addition, the presence of many type casts spread throughout the residual code obscures its functionality and complicates manual inspection. As a simple means of reducing the number of type casts, we are interested in changing the types of local variables and formal parameters to be more specific when the objects that they reference are known to have a more specific type. Although easily incorporated into the minimal partial evaluator described in Chapter 8, such a transformation would be difficult to implement in JSpec due to our use of C as an intermediate language (as described in Chapter 10); Java-specific code that inspects the run-time state of objects is needed to implement this transformation.

13.2.4 Program size optimization

When deploying a program as mobile code or in an embedded system, it can be useful to optimize the size of the program. In the context of Java, such optimization is often achieved using compression techniques and by removing redundant information [24, 136, 138]. To reduce the size of programs independently of what can be achieved with compression techniques, we are interested in applying partial evaluation to configure programs to only perform certain tasks. Ideally, code size reduction due to partial evaluation would be complementary to code size reduction obtained using compression. Nevertheless, certain issues pertaining to controlling partial evaluation would have to be worked out before this approach would be practical, since simply applying partial evaluation often increases program size due to loop unrolling or the generation of many specialized method variants.

Java programs can be represented in different ways, depending on the target domain. For a maximal effect, this variation must to be taken into account when deciding what parts of a Java program to target with a partial evaluator. In the standard Java class file format, roughly 20% of the space is taken up by bytecode [7]; the remaining space is taken up by class declarations and symbolic information for the linker. Thus, partial evaluation should be targeted to eliminate fields, method and even entire classes from the program; to this end, we plan to use a combination of partial evaluation and class-hierarchy specialization [155] (see Chapter 3). However, when used in a low-end embedded system such as JavaCard [148], almost 75% of the space is taken up by bytecode [38], and significant amount of space can be saved reducing the size of individual methods.

13.3 Continued JSpec Development

JSpec is a large and complex tool, and we have identified and described numerous possible improvements: the speed of the analysis phase could be improved by adding Java-specific functionality to the Tempo core; the performance of specialized programs could be improved by adding common subexpression elimination and object inlining as post-specialization optimizations; the set of programs that can be specialized would be increased by adding support for Java features such as reflection, exceptions and multi-threading. In addition to these features, we are interested in adding support for run-time specialization of Java bytecode, which we will concentrate on in this section.

Currently, run-time specialization in JSpec generates binary code for execution in the Harissa environment by using the Tempo run-time specializer [43]. With this approach the platform-independence advantages of Java are lost; although both the Tempo run-time specializer and Harissa are constructed with portability in mind, Java execution environments are available for far more platforms than these two tools. To allow Java bytecode to be specialized at run time in any standard Java virtual machine, a run-time specializer must execute within a standard Java environment and use Java dynamic class loading to install new code into the program. Such a run-time specializer could probably be constructed by porting the existing run-time specializer to Java and extending it with automatic back-translation from intermediate C to Java bytecode.

The implementation of such a run-time specializer would be a considerable engineering task. Furthermore, it is not obvious how the specialized intermediate C code can be back-translated into specialized Java code that can be dynamically loaded. We express a specialized program as an AspectJ aspect; there is no means for dynamically loading an aspect into a Java program, since an aspect may modify the definition of classes already loaded into the program. To overcome this limitation, our approach for generating specialized programs must either be redesigned, or the Java virtual machine must be extended to permit dynamic modification of classes. In fact, a Java virtual machine can be redesigned to allow classes to be dynamically updated, as shown by Gupta et al. [71].

Dynamic loading of code into a JIT or a dynamic compiler has the disadvantage that the code must be compiled before it will be efficient, but has the advantage that once compiled this code is likely to be better optimized than the binary code generated by a template-based approach such as the one used in the Tempo run-time specializer. An experimental study will be needed to assess the overhead due to JIT or dynamic compilation, to determine when run-time specialization can be worthwhile.

13.4 Partial Evaluation and Software Engineering

To improve the utility of partial evaluation as a software engineering technique, we are interested in improving the tool support for using partial evaluation, further enhancing the specialization pattern approach, and applying partial evaluation within the domain of software components.

13.4.1 Tool support for partial evaluation

To facilitate the use of partial evaluation to specialize object-oriented programs, tools can be developed to assist the user in applying partial evaluation. In the context of object-oriented programs, we are interested in developing a "specialization assistant" that could allow a target program slice and the static parts of its context to be identified in a UML diagram representation of the entire program. The information indicated by the user can then be used to automatically generate specialization classes reflecting the specialization scenario indicated by the user. For a more complete integration of specialization into the program, a graphical syntax could also be developed for specialization classes, to allow them to be integrated into UML diagrams.

13.4.2 Specialization patterns

With a formal definition of design patterns, it is possible that specialization patterns also could be formalized; such a formalization could allow automatic application of specialization patterns, and user guidance of the specialization process could be greatly simplified. For example, when the source language has support for design patterns [23, 74, 95] or when the program is developed using a CASE tool that supports design patterns [44, 108], specialization classes could be automatically generated for each use of a design pattern. These specialization classes would then precisely define the specialization capabilities of the resulting program.

13.4.3 Partial evaluation for software components

Partial evaluation can be applied within the domain of software components in two largely orthogonal ways: as a means to optimize intercomponent interaction, and as a means to configure each component to a specific functionality.

The Java Beans component architecture is defined using standard Java constructs under certain constraints. Just as a framework can systematically introduce specific overheads, the Java Beans component architecture also introduces overheads into programs. We are interested in applying partial evaluation to automatically optimize away these overheads. For example, the standard Java Beans event model can be specialized with an overall effect similar to the specialization of the strategy design pattern (see Chapter 7). Concretely, we aim to completely automate the partial evaluation process for the specific case of Java Beans, by automatically generating specialization classes.

To encourage the use of partial evaluation as a software engineering tool, we are interested in giving guidelines for creating automatically configurable program parts. It is likely that a software component is an appropriate unit of granularity: a component can be written in a generic and parameterized way so that partial evaluation can be used as an automatic tool to configure it to a specific use. This approach would permit automatic generation of a whole family of components from a single implementation. For a maximal benefit, support for partial evaluation must be integrated into the component architecture, to permit automatic configuration at the component level according to how components are combined into a complete program [142].

13.5 Summary

We aim to further explore the concepts of object-oriented partial evaluation, to improve the implementation of JSpec, and to investigate software engineering aspects of using partial evaluation. In addition to proving the correctness of object-oriented partial evaluation to ensure its correctness, our overall goal is to facilitate the use of object-oriented partial evaluation and to extend the set of programs and specialization opportunities that can be targeted using partial evaluation.

Chapter 14

Conclusion

The are three main parts to our contribution: we give a general definition of partial evaluation for object-oriented languages, we develop a partial evaluator for realistic Java programs, and we define an approach to integrating partial evaluation with software engineering. Together, these three parts form a complete approach to partial evaluation for object-oriented languages.

14.1 Object-Oriented Partial Evaluation

Partial evaluation constitutes a minimalistic, formalized description of how complete object-oriented programs can be specialized; it is driven by a uniform analysis that controls a set of well-defined transformations. The principles here presented describe partial evaluation in terms of features found in any object-oriented language; the more specific features of most languages remain to be investigated. To define partial evaluation for object-oriented languages, we have given a conceptual account of its effect, a formalization of its semantics, and a description of the features needed to specialize realistic programs. Compared to other automatic specialization approaches for object-oriented languages, partial evaluation offers more aggressive optimizations and additional user control, but requires explicit user intervention for successful specialization of most programs.

In terms of understanding, the relation between partial evaluation and inheritance has been made clear: partial evaluation specializes programs, not classes, but may, when convenient, employ inheritance to express the result of specialization. Likewise, the intuitive similarity between partial evaluation and aspect-oriented programming has been made clear: an aspect language may be used to control partial evaluation, and the result of specializing an object-oriented program is an aspect of the program. This similarity has proven fruitful for object-oriented partial evaluation, since aspect-oriented programming has proven to be essential in clearly and concisely expressing the result of specializing an object-oriented program.

14.2 Partial Evaluation for Java

We have presented our implementation of a complete partial evaluator for realistic Java programs, named JSpec. JSpec incorporates the principles and techniques of object-oriented partial evaluation, and documents the feasibility of this technique. In terms of efficiency, Java sports a hitherto unmatched plurality in execution environments, and partial evaluation provides execution speed advantages across most kinds of environments.

JSpec has been conceived as a direct extension of the Tempo partial evaluator for C. We have shown this implementation approach to be feasible in practice; a limited amount of implementation work has directly produced a Java partial evaluator with features normally found only in mature partial evaluators. However, the complexity of this layered implementation combined with the difficulty inherent in implementing Java-specific features has led to the conclusion that implementing basic Java-specific features directly in Tempo, while not necessary, would have been advantageous.

14.3 Software Engineering

To facilitate the use of partial evaluation, we have in this document given concrete guidelines for integrating partial evaluation with the software engineering process. Taking partial evaluation into account during the development of a program ensures that the finished program will specialize well. Specialization classes offer a convenient means for specifying specialization during program development. Specialization patterns link partial evaluation experience with specific program designs.

We believe that the specialization pattern approach is essential for widespread use of partial evaluation. Partial evaluation is difficult to use, and a means for communicating experience in applying partial evaluation is needed; specialization patterns convey this information in a convenient form. In the specific case of object-oriented languages, design patterns offer advantages in terms of program design, and specialization patterns extend these advantages by using the design patterns to guide specialization of the program.

14.4 Final Remarks

This document complements the existing work in partial evaluation for functional, logical and imperative languages: it defines partial evaluation for the object-oriented programming paradigm. Nonetheless, much work remains in investigating the nature of partial evaluation for object-oriented languages. This document is intended as a source of inspiration for future research avenues rather than as a final treatment of the subject. However, the true challenge for the future lies in fulfilling the promise of partial evaluation: that partial evaluation become a standard program development utility in the programmer's toolbox.

Appendix A

Implementation and Formalization Details

This appendix first describes implementation details (Section A.1), and then describes details in the formalization of object-oriented partial evaluation (Section A.2).

A.1 Example Implementation Details

Figure A.1 defines the underlying data structures used for the iterator design pattern example shown in Chapter 4. The class `Array` functions as a wrapper for a standard Java array, and in addition defines a method for obtaining an iterator. The iterator is defined using the class `ArrayIterator`; it uses local fields to store information about the current element and `Array` object that it iterates over.

Figure A.2 shows the complete source code of the Scheme self-interpreter used as an example of frozen closure specialization in Chapter 9. It is used as follows:

```
(interpret '() '(apply (lambda (x y) (+ (var x) (var y)))
                       (const 1) (const 2)))
```

which simply produces

```
3
```

The main function `interpret` dispatches based on the syntactic form, and the environment is represented as a list of pairs of variable name and computed value.

A.2 Formalization Details

Figure A.3 defines the auxiliary definitions used in Figure 8.7. The function *no-bt* removes all binding-time annotations from an expression. The function *build-env* builds a type environment for analysis of a method. The function

```
class Array implements MinimalCollection {
  Object []elements;
  int size;
  Array(int size) {
    this.size = size;
    this.elements = new Object[size];
  }
  public Object get(int n) { return elements[n]; }
  public int getSize() { return size; }
  Iterator iterator() {
    return new ArrayIterator( this );
  }
  ... other methods for implementing MinimalCollection ...
}
class ArrayIterator implements Iterator {
  Array array;
  int current, max;
  ArrayIterator( Array a ) {
    this.array = a;
    this.current = 0;
    this.max = a.getSize();
  }
  boolean hasNext() { return current<max; }
  Object next() { return array.get(current++); }
}
```

Figure A.1: Relevant parts of `Array` and its iterator `ArrayIterator`.

class-bt returns the binding-time of a class, and the function *field-bt* returns the binding-time of a given field of a class. Last, the function *bt-signature* returns the binding-time signature of a method, and the functions *static-bt* and *param-bt* are auxiliary functions used in the definition of *bt-signature*.

Figure A.4 defines the improved reduction rules for specialization by 2EFJ reduction. These reduction rules reduce a tuple consisting of the pending methods and an expression into a specialized expression and a set of specialized methods. The pending methods are identified by their name, class, and the static values or residual expressions for which they are being specialized. The rules are mostly equivalent to the standard 2EFJ reduction rules, except that there are two rules for reduction of methods with a dynamic self. The rule I2-R-D-INVK-1 reduces method invocations for which there is a matching, pending specialized method; a call to the pending method is simply residualized. The rule I2-R-D-INVK-2 is similar to the standard rule for reducing a method invocation with a dynamic self, except that the methods that are to be specialized must be added to the set of pending methods. A number of auxiliary rules are used, defined in Figure A.5; the function *exists* checks whether an appropriate residual function exists in the set of pending methods; the function *cache-entries* converts a set of specialized methods into a set of pending methods, and is used to avoid unnecessary respecialization of methods.

```
; interpret-lambda: (environment,[parameter],exp) -> ([value] -> value)
(define interpret-lambda
  (lambda (env formals body)
    (lambda (arguments) (interpret (combine formals arguments env) body))))

; interpret-apply: (environment,exp,[exp]) -> value
(define interpret-apply
  (lambda (env e0 args)
    ((interpret env e0) (map (lambda (a) (interpret env a)) args))))

; interpret-plus: (environment,exp,exp) -> value
(define interpret-plus
  (lambda (env e0 e1)
    (+ (interpret env e0) (interpret env e1))))

; interpret-var: (environment,name) -> value
(define interpret-var
  (lambda (env name)
    (lookup env name)))

; interpret-const: (environment,value) -> value
(define interpret-const
  (lambda (env val)
    val))

; interpret: (environment,exp) -> value
(define interpret
  (lambda (env exp)
    (let ((syntax (car exp))
          (rest (cdr exp)))
      (cond ((eq? syntax 'lambda)
             (interpret-lambda env (car rest) (car (cdr rest))))
            ((eq? syntax 'apply)
             (interpret-apply env (car rest) (cdr rest)))
            ((eq? syntax '+)
             (interpret-plus env (car rest) (car (cdr rest))))
            ((eq? syntax 'var)
             (interpret-var env (car rest)))
            ((eq? syntax 'const)
             (interpret-const env (car rest)))
            (#t (error "bad syntax:" syntax))))))

; combine: ([var],[value],environment) -> environment
(define combine
  (lambda (vars values env)
    (if (null? vars)
        env
        (cons (cons (car vars) (car values))
              (combine (cdr vars) (cdr values) env)))))

; lookup: (environment,var) -> value
(define lookup
  (lambda (env var)
    (cond ((null? env) (error "undefined variable" var))
          ((eq? (car (car env)) var) (cdr (car env)))
          (#t (lookup (cdr env) var)))))
```

Figure A.2: Functional language self-interpreter in Scheme.

$$\textit{no-bt}(e) = e \text{ with all binding-time annotations and } \texttt{lift} \text{ operators removed}$$

$$\textit{build-env}((\texttt{x}_1, \ldots, \texttt{x}_\texttt{n}), T_0, (T_1, \ldots, T_n)) = [\texttt{this} \mapsto T_0, \texttt{x}_1 \mapsto T_1, \ldots, \texttt{x}_\texttt{n} \mapsto T_n]$$

$$\frac{\textit{CT}(\texttt{C}) = \texttt{class C extends D} \{\ldots;\ \texttt{K}\ \ldots\}}{\textit{class-bt}(\texttt{C}) = S_N}$$

$$\frac{\textit{CT}(\texttt{C}) = \texttt{class C extends D} \{\ldots;\ \underline{\texttt{K}}\ \ldots\}}{\textit{class-bt}(\texttt{C}) = D}$$

$$\frac{\textit{class-bt}(\texttt{C}) = D}{\textit{field-bt}(\texttt{C}, \texttt{f}) = D} \qquad \frac{\textit{class-bt}(\texttt{C}) = S_N \quad \texttt{T f} \in \textit{fields}(\texttt{C}), \texttt{T} \notin \{\texttt{int}, \texttt{boolean}\}}{\textit{field-bt}(\texttt{C}, \texttt{f}) = S_N}$$

$$\frac{\textit{class-bt}(\texttt{C}) = S_N \quad \texttt{T f} \in \textit{fields}(\texttt{C}), \texttt{T} \in \{\texttt{int}, \texttt{boolean}\}}{\textit{field-bt}(\texttt{C}, \texttt{f}) = S_L}$$

$$\frac{\begin{array}{c}\textit{CT}(\texttt{C}) = \texttt{class C extends D \{ ...M}_1\texttt{...M}_k\texttt{ \}} \\ \texttt{M}_i = \texttt{T m}(P)\ \{\ \texttt{return}\ e\ \} \qquad \textit{mtype}(\texttt{m}, \texttt{C}) = T_0.(T_1, \ldots, T_n) \mapsto T_R \\ T_0' = \textit{class-bt}(\texttt{C}) \qquad T_i' = \textit{param-bt}(\#i(P), T_i), i \neq 0 \qquad T_R' = \textit{static-bt}(T_R)\end{array}}{\textit{bt-signature}(\texttt{C}, \texttt{m}) = T_0'.(T_1', \ldots, T_n') \mapsto T_R'}$$

$$\frac{\begin{array}{c}\textit{CT}(\texttt{C}) = \texttt{class C extends D \{ ...M}_1\texttt{...M}_k\texttt{ \}} \\ \texttt{M}_i = \texttt{T m}(P)\ \{\ \underline{\texttt{return}}\ e\ \} \qquad \textit{mtype}(\texttt{m}, \texttt{C}) = T_0.(T_1, \ldots, T_n) \mapsto T_R \\ T_0' = \textit{class-bt}(\texttt{C}) \qquad T_i' = \textit{param-bt}(\#i(P), T_i), i \neq 0 \qquad T_R' = D\end{array}}{\textit{bt-signature}(\texttt{C}, \texttt{m}) = T_0'.(T_1', \ldots, T_n') \mapsto T_R'}$$

$$\frac{\begin{array}{c}\textit{CT}(\texttt{C}) = \texttt{class C extends D \{ ...M}_1\texttt{...M}_k\texttt{ \}} \\ \texttt{m} \text{ not defined in } \texttt{M}_1, \ldots, \texttt{M}_k\end{array}}{\textit{bt-signature}(\texttt{C}, \texttt{m}) = \textit{bt-signature}(\texttt{D}, \texttt{m})}$$

$$\frac{T \in \{\texttt{int}, \texttt{boolean}\}}{\textit{static-bt}(T) = S_L} \qquad \frac{T \notin \{\texttt{int}, \texttt{boolean}\}}{\textit{static-bt}(T) = S_N}$$

$$\frac{T \in \{\texttt{int}, \texttt{boolean}\}}{\textit{param-bt}(\texttt{x}, T) = S_L} \qquad \frac{T \notin \{\texttt{int}, \texttt{boolean}\}}{\textit{param-bt}(\texttt{x}, T) = S_N} \qquad \textit{param-bt}(\underline{\texttt{x}}, T) = D$$

Figure A.3: Auxiliary definitions for Figure 8.7.

Figure A.6 shows the operator Δ-reduction rules for EFJ. Each variant of the Δ rules reduces two arguments, either integers or booleans, into a single value. Intuitively, each operator defined by the Δ rules corresponds to the equivalent Java operator.

Each EFJ reduction rule from Figure 8.5 is extended to pass the set of pending methods to be specialized to the evaluation of its sub-expressions and to collect residual specialized methods, in the same way as the 2EFJ reduction rules below. The 2EFJ version of an EFJ rule (R-X) is named (I2-R-S-X), giving the rules (I2-R-S-New), (I2-R-S-Field), (I2-R-S-Invk), (I2-R-S-Cast), (I2-R-S-Cond), and (I2-R-S-OP).

$$\frac{\mathrm{vn} = \mathit{name}(\mathtt{x})}{(N, \underline{\mathtt{x}}) \longrightarrow (\mathit{build\text{-}var}(\mathrm{vn}), \emptyset)} \qquad \text{(I2-R-D-Var)}$$

$$\frac{(N, e) \longrightarrow (r, M)}{(N, \mathtt{lift}(e)) \longrightarrow (\mathit{build\text{-}const}(r), M)} \qquad \text{(I2-R-Lift)}$$

$$\frac{(N, e_i) \longrightarrow (r_i, M_i) \qquad \mathrm{cn} = \mathit{name}(\mathtt{C})}{(N, \underline{\mathtt{new\ C}}(e_1, \ldots, e_n)) \longrightarrow (\mathit{build\text{-}new}(\mathrm{cn}, (r_1, \ldots, r_n)), \cup M_i)} \qquad \text{(I2-R-D-New)}$$

$$\frac{(N, e) \longrightarrow (r, M) \qquad \mathrm{fn} = \mathit{name}(\mathtt{f})}{(N, e\underline{.\mathtt{f}}_{\{\mathtt{C_1}, \ldots, \mathtt{C_k}\}}) \longrightarrow (\mathit{build\text{-}field\text{-}lookup}(r, \mathrm{fn}), M)} \qquad \text{(I2-R-D-Field)}$$

$$\frac{(N, e) \longrightarrow (r, M) \qquad \mathrm{cn} = \mathit{name}(\mathtt{C})}{(N, \underline{(\mathtt{C})}e) \longrightarrow (\mathit{build\text{-}cast}(r, \mathrm{fn}), M)} \qquad \text{(I2-R-D-Cast)}$$

$$\frac{(N, e_i) \longrightarrow (r_i, M_i)}{(N, e_0\underline{?}e_1\underline{:}e_2) \longrightarrow (\mathit{build\text{-}if}(r_0, r_1, r_2), \cup M_i)} \qquad \text{(I2-R-D-Cond)}$$

$$\frac{(N, e_i) \longrightarrow (r_i, M_i) \qquad \mathrm{on} = \mathit{name}(\mathtt{OP})}{(N, e_0\ \underline{\mathtt{OP}}\ e_1) \longrightarrow (\mathit{build\text{-}op}(\mathrm{on}, r_0, r_1), \cup M_i)} \qquad \text{(I2-R-D-OP)}$$

$$\frac{\begin{array}{c}(N, e_i) \longrightarrow (r_i, M_i) \quad N' = \mathit{cache\text{-}entries}(\cup M_i) \\ \mathit{exists}(N', \{\mathtt{C_1}, \ldots, \mathtt{C_k}\}, \mathtt{m}, (r_1, \ldots, r_n)) = \{\mathrm{mn}\}\end{array}}{\begin{array}{l}(N, e_0\underline{.\mathtt{m}}_{\{\mathtt{C_1}, \ldots, \mathtt{C_k}\}}(e_1, \ldots, e_n)) \\ \qquad \longrightarrow (\mathit{build\text{-}invoke}(\mathtt{m}, \mathrm{mn}, r_0, (r_1, \ldots, r_n)), \cup M_i)\end{array}} \qquad \text{(I2-R-D-Invk-1)}$$

$$\frac{\begin{array}{c}(N, e_i) \longrightarrow (r_i, M_i) \quad N' = \mathit{cache\text{-}entries}(\cup M_i) \\ \mathit{exists}(N', \{\mathtt{C_1}, \ldots, \mathtt{C_k}\}, \mathtt{m}, (r_1, \ldots, r_n)) = \emptyset \qquad \mathrm{mn} = \mathit{new\text{-}name}(N', \mathtt{m}) \\ N'' = N' \cup \{(\mathtt{C_i}, \mathtt{m}, \mathrm{mn}, (r_1, \ldots, r_n))\}_{i \in 1 \ldots k} \qquad \mathit{mbody}(\mathtt{C_i}, \mathtt{m}) = (\ldots, d_i) \\ \alpha_i = \mathit{build\text{-}subst}(\mathtt{C_i}, \mathtt{m}, (r_1, \ldots, r_n)) \quad (N'', \alpha_i d_i) \longrightarrow (d'_i, M'_i) \\ m_i = \mathit{build\text{-}method'}(\mathtt{C_i}, \mathtt{m}, \mathrm{mn}, d'_i) \quad N''' = (\cup M_i) \cup (\cup M'_i) \cup \{m_1, \ldots, m_k\}\end{array}}{(N, e_0\underline{.\mathtt{m}}_{\{\mathtt{C_1}, \ldots, \mathtt{C_k}\}}(e_1, \ldots, e_n)) \longrightarrow (\mathit{build\text{-}invoke}(\mathtt{m}, \mathrm{mn}, r_0, (r_1, \ldots, r_n)), N''')}$$

(I2-R-D-Invk-2)

Pending new methods N: $\{(((\mathsf{Val}|\mathsf{Exp}) \times \ldots \times (\mathsf{Val}|\mathsf{Exp})), \mathsf{Class}, \mathsf{Method}, \mathsf{Method})\}$
Method cache M:
$\{(((\mathsf{Val}|\mathsf{Exp}) \times \ldots \times (\mathsf{Val}|\mathsf{Exp})), \mathsf{Class}, \mathsf{Type}, \mathsf{Method}, \mathsf{Method}, (\mathsf{Var} \times \ldots \times \mathsf{Var}), \mathsf{Exp})\}$

Figure A.4: Specialization as two-level execution, with a method cache.

$$\frac{\mathcal{T} = \mathit{signature}(\{\mathtt{C}_1, \ldots, \mathtt{C}_\mathtt{k}\}, \mathtt{m}) \quad \exists \mathrm{mn} \in \mathit{names}(N) : \xi(N, \mathtt{C}_1, \ldots, \mathtt{C}_\mathtt{k}\}, \mathrm{mn}, (r_1, \ldots, r_n), \mathcal{T})}{\mathit{exists}(N, \{\mathtt{C}_1, \ldots, \mathtt{C}_\mathtt{k}\}, \mathtt{m}, (r_1, \ldots, r_n)) = \{\mathrm{mn}\}}$$

$$\frac{\mathcal{T} = \mathit{signature}(\{\mathtt{C}_1, \ldots, \mathtt{C}_\mathtt{k}\}, \mathtt{m}) \quad \neg\exists \mathrm{mn} \in \mathit{names}(N) : \xi(N, \{\mathtt{C}_1, \ldots, \mathtt{C}_\mathtt{k}\}, \mathrm{mn}, (r_1, \ldots, r_n), \mathcal{T})}{\mathit{exists}(N, \mathtt{C}_1, \ldots, \mathtt{C}_\mathtt{k}\}, \mathtt{m}, (r_1, \ldots, r_n)) = \emptyset}$$

$$\begin{aligned} \xi(N, \mathcal{C}, \mathtt{m}, (r_1, \ldots, r_n), \mathcal{T}) = \; & \forall \mathtt{C} \in \mathcal{C} \exists ((v_1, \ldots, v_p), \mathtt{C}, \mathtt{m}, \mathrm{mn}) \in N : \\ & \mathit{match}(\mathcal{T}, (r_1, \ldots, r_n), (v_1, \ldots, v_p)) \end{aligned}$$

$$\frac{}{\mathit{match}(T_0.(()) \mapsto T_R, (\ldots), ())}$$

$$\frac{r = v \quad \mathit{match}(T_0.((\ldots)) \mapsto T_R, (\ldots), (\ldots))}{\mathit{match}(T_0.((S_\gamma, \ldots)) \mapsto T_R, (r, \ldots), (v, \ldots))}$$

$$\frac{\mathit{match}(T_0.((\ldots)) \mapsto T_R, (\ldots), (v, \ldots))}{\mathit{match}(T_0.((D, \ldots)) \mapsto T_R, (r, \ldots), (v, \ldots))}$$

$$\mathit{cache\text{-}entries}(M) = \gamma(M), \gamma(\mathtt{C}, \mathtt{m}, T, \mathrm{mn}, (v_1, \ldots, v_p), (x_1, \ldots, x_q), e) = (\mathtt{C}, \mathtt{m}, \mathrm{mn}, (v_1, \ldots, v_p))$$

Figure A.5: Auxiliary definitions for Figure A.4.

$$\begin{array}{lcl}
\Delta_{\mathtt{+}}(v_1, v_2) & = & v_1 + v_2,\ v_1, v_2 \in \{0, 1, -1, \ldots\} \\
\Delta_{\mathtt{-}}(v_1, v_2) & = & v_1 - v_2,\ v_1, v_2 \in \{0, 1, -1, \ldots\} \\
\Delta_{\mathtt{*}}(v_1, v_2) & = & v_1 * v_2,\ v_1, v_2 \in \{0, 1, -1, \ldots\} \\
\Delta_{\mathtt{/}}(v_1, v_2) & = & v_1 / v_2,\ v_1, v_2 \in \{0, 1, -1, \ldots\} \\
\Delta_{<}(v_1, v_2) & = & \mathtt{true} \text{ if } v_1 < v_2,\ v_1, v_2 \in \{0, 1, -1, \ldots\} \\
 & = & \mathtt{false} \text{ otherwise},\ v_1, v_2 \in \{0, 1, -1, \ldots\} \\
\Delta_{>}(v_1, v_2) & = & \mathtt{true} \text{ if } v_1 > v_2,\ v_1, v_2 \in \{0, 1, -1, \ldots\} \\
 & = & \mathtt{false} \text{ otherwise},\ v_1, v_2 \in \{0, 1, -1, \ldots\} \\
\Delta_{\mathtt{==}}(v_1, v_2) & = & \mathtt{true} \text{ if } v_1 = v_2,\ v_1, v_2 \in \{0, 1, -1, \ldots\} \\
 & = & \mathtt{false} \text{ otherwise},\ v_1, v_2 \in \{0, 1, -1, \ldots\} \\
\Delta_{\mathtt{==}}(b_1, b_2) & = & \mathtt{true} \text{ if } b_1 = b_2,\ b_1, b_2 \in \{\mathtt{true}, \mathtt{false}\} \\
 & = & \mathtt{false} \text{ otherwise},\ b_1, b_2 \in \{\mathtt{true}, \mathtt{false}\} \\
\Delta_{\mathtt{\&\&}}(v_1, v_2) & = & \mathtt{true} \text{ if } v_1 = v_2 = \mathtt{true} \\
 & = & \mathtt{false} \text{ otherwise} \\
\Delta_{\mathtt{||}}(v_1, v_2) & = & \mathtt{false} \text{ if } v_1 = v_2 = \mathtt{false} \\
 & = & \mathtt{true} \text{ otherwise}
\end{array}$$

Figure A.6: Delta reduction rules for EFJ.

Bibliography

[1] M. Abadi and L. Cardelli. *A Theory of Objects.* Springer-Verlag, N.Y., USA, 1996.

[2] O. Agesen, J. Palsberg, and M.I. Schwartzbach. Type inference of SELF. In ECOOP'93 [55], pages 247–267.

[3] G. Aigner and U. Hölzle. Eliminating virtual function calls in C++ programs. In *Proceedings of the European Conference on Object-oriented Programming (ECOOP'96)*, volume 1098 of *Lecture Notes in Computer Science*, pages 142–166, Linz, Austria, July 1996. Springer-Verlag.

[4] B. Alpern, C.R. Attanasio, A. Cocchi, D. Lieber, S. Smith, T. Ngo, J.J. Barton, S.F. Hummel, J.C. Sheperd, and M. Mergen. Implementing Jalapeño in Java. In Meissner [109], pages 314–324.

[5] L.O. Andersen. Binding-time analysis and the taming of C pointers. In PEPM'93 [126], pages 47–58.

[6] L.O. Andersen. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, Computer Science Department, University of Copenhagen, May 1994. DIKU Technical Report 94/19.

[7] D.N. Antonioli and M. Pilz. Analysis of the Java class file format. Technical Report ifi-98.04, Department of Computer Science, University of Zurich, April 1998.

[8] K. Asai. Binding-time analysis for both static and dynamic expressions. In A. Cortesi and G. Filé, editors, *Static Analysis Symposium*, volume 1694 of *Lecture Notes in Computer Science*, pages 117–133. Springer-Verlag, 1999.

[9] K. Asai, H. Masuhara, and A. Yonezawa. Partial evaluation of call-by-value lambda-calculus with side-effects. In PEPM'97 [127], pages 12–21.

[10] *Proceedings of the 15th IEEE International Conference on Automated Software Engineering (ASE 2000)*, Grenoble, France, September 2000. IEEE Computer Society Press.

[11] J. Auslander, M. Philipose, C. Chambers, S. Eggers, and B.N. Bershad. Fast, effective dynamic compilation. In PLDI'96 [129], pages 149–159.

[12] R. Baier, R. Glück, and R. Zöchling. Partial evaluation of numerical programs in Fortran. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'94)*, pages 119–132, Orlando, FL, USA, June 1994. Technical Report 94/9, University of Melbourne, Australia.

[13] Z. Benaissa and A. Tolmach. Interface-directed partial evaluation. Technical Report CSE-99-010, Department of Computer Science and Engineering, Oregon Graduate Institute of Science & Technology, September 1999.

[14] A.A. Berlin. Partial evaluation applied to numerical computation. In *ACM Conference on Lisp and Functional Programming*, pages 139–150, Nice, France, 1990. ACM Press.

[15] B.N. Bershad, T.E. Anderson, E.D. Lazowska, and H.M. Levy. Lightweight remote procedure call. *ACM Transactions on Computer Systems*, 8(1):37–55, February 1990.

[16] P. Bertelsen. Binding-time analysis for a JVM core language, April 1999. Unpublished note; available from `http://www.dina.kvl.dk/~pmb`.

[17] L. Birkedal and M. Welinder. Partial evaluation of Standard ML. Master's thesis, Computer Science Department, University of Copenhagen, 1993. Research Report 93/22.

[18] B. Blanchet. Escape analysis for object-oriented languages: application to Java. In Meissner [109], pages 20–34.

[19] D.G. Bobrow, L.G. DeMichel, R.P. Gabriel, S.E. Keene, G. Kiczales, and D.A. Moon. Common Lisp Object System specification X3J13. *ACM SIGPLAN Notices*, 23(9), 1988.

[20] P. Boinot, R. Marlet, Noyé J., G. Muller, and C. Consel. A declarative approach for designing and developing adaptive components. In ASE'00 [10].

[21] A. Bondorf. *Self-Applicable Partial Evaluation.* PhD thesis, DIKU, University of Copenhagen, Denmark, 1990. Revised version: DIKU Report 90/17.

[22] A. Bondorf and J. Jørgensen. Efficient analyses for realistic off-line partial evaluation. *Journal of Functional Programming*, 3(3):315–346, July 1993.

[23] J. Bosch. Design patterns as language constructs. *Journal of Object-Oriented Programming*, November 1996.

[24] Q. Bradly, R.N. Horspool, and J. Vitek. Jazz: An efficient compressed format for Java archive files. In *Proceedings of CASCON'98*, pages 294–302, Toronto, Ontario, November 1998.

[25] M. Braux and J. Noyé. Towards partially evaluating reflection in Java. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'00)*, Boston, MA, USA, January 2000. ACM Press.

[26] Z. Budimlic and K. Kennedy. Optimizing Java: theory and practice. *Concurrency: Practice and Experience*, 9(6):445–463, June 1997.

[27] M.A. Bulyonkov and D.V. Kochetov. Practical aspects of specialization of Algol-like programs. In Danvy et al. [46], pages 17–32.

[28] B. Calder and D. Grunwald. Reducing indirect function call overhead in C++ programs. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages (POPL'94)*, pages 397–408. ACM Press, 1994.

[29] T. Cargill. *C++ programming style.* Addison-Wesley professional computing series. Addison-Wesley, Reading, MA, USA, 1992.

[30] C. Chambers. Object-oriented multi-methods in Cecil. In Madsen [101], pages 33–56.

[31] C. Chambers. Predicate classes. In ECOOP'93 [55], pages 268–296.

[32] C. Chambers. The Cecil language specification and rationale: Version 2.1. Available from `http://www.cs.washington.edu/research/projects/cecil`, December 1997.

[33] C. Chambers and D. Ungar. Customization: Optimizing compiler technology for SELF, A dynamically-typed object-oriented programming language. In Bruce Knobe, editor, *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation (PLDI '89)*, pages 146–160, Portland, OR, USA, June 1989. ACM Press.

[34] R. Chatterjee, B.G. Ryder, and W. Landi. Complexity of concrete type-inference in the presence of exceptions. In C. Hankin, editor, *Programming Languages and Systems, 7th European Symposium on Programming (ESOP'98)*, volume 1381 of *Lecture Notes in Computer Science*, pages 57–74, Lisbon, Portugal, March 1998. Springer-Verlag.

[35] W.H. Cheung and A. Loong. Exploring issues of operating systems structuring: from microkernel to extensible systems. *ACM Operating Systems Reviews*, 29(4):4–16, October 1995.

[36] J. Choi, D. Grove, M. Hind, and V. Sarkar. Efficient and precise modeling of exceptions for the analysis of Java programs. In *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, volume 24.5 of *Software Engineering Notes (SEN)*, pages 21–31, N.Y., USA, September 1999. ACM Press.

[37] J. Choi, M. Gupta, M. Serrano, Vugranam C. Sreedhar, and Sam Midkiff. Escape analysis for Java. In Meissner [109], pages 1–19.

[38] L. Clausen, U.P. Schultz, C. Consel, and G. Muller. Java bytecode compression for low-end embedded systems. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(3):471–489, May 2000.

[39] C. Consel. Polyvariant binding-time analysis for applicative languages. In PEPM'93 [126], pages 145–154.

[40] C. Consel. A tour of Schism: a partial evaluation system for higher-order applicative languages. In PEPM'93 [126], pages 66–77.

[41] C. Consel, L. Hornof, F. Noël, J. Noyé, and E.N. Volanschi. A uniform approach for compile-time and run-time specialization. In Danvy et al. [46], pages 54–72.

[42] C. Consel and S.C. Khoo. On-line and off-line partial evaluation: Semantic specifications and correctness proofs. *Journal of Functional Programming*, 5(4):461–500, October 1995.

[43] C. Consel and F. Noël. A general approach for run-time specialization and its application to C. In POPL'96 [132], pages 145–156.

[44] A. Cornils and G. Hedin. Statically checked documentation with design patterns. In R. Mitchell, J. Jezequel, J. Bosch, B. Meyer, A.C. Wills, and M. Woodman, editors, *Proceedings of TOOLS Europe 33*, pages 419–431, Mt. St. Michel, France, June 2000. IEEE Computer Society Press.

[45] O. Danvy. Type-directed partial evaluation. In POPL'96 [132], pages 242–257.

[46] O. Danvy, R. Glück, and P. Thiemann, editors. *Partial Evaluation, International Seminar, Dagstuhl Castle*, number 1110 in Lecture Notes in Computer Science, February 1996.

[47] O. Danvy and U.P. Schultz. Lambda-dropping: Transforming recursive equations into programs with block structure. *Theoretical Computer Science (TCS)*, 2000. to appear in TCS Volume 248/1-2.

[48] J. Dean, C. Chambers, and D. Grove. Selective specialization for object-oriented languages. In PLDI'95 [128], pages 93–102.

[49] J. Dean, G. DeFouw, D. Grove, V. Litvinov, and C. Chambers. Vortex: an optimizing compiler for object-oriented languages. In OOPSLA'96 [122], pages 93–100.

[50] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In Walter G. Olthoff, editor, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'95)*, volume 952 of *Lecture Notes in Computer Science*, pages 77–101, Århus, Denmark, August 1995. Springer-Verlag.

[51] D. Detlefs and O. Agesen. Inlining of virtual methods. In ECOOP'99 [56], pages 258–278.

[52] K. Driesen and U. Hölzle. The direct cost of virtual function calls in C++. In OOPSLA'96 [122], pages 306–323.

[53] D. Dussart, R. Heldal, and J. Hughes. Module-sensitive program specialization. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation (PLDI'97)*, pages 206–214, Las Vegas, NV, USA, June 1997.

[54] *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'00)*, volume 1850 of *Lecture Notes in Computer Science*, Cannes, France, 2000. Springer-Verlag.

[55] *Proceedings of the European Conference on Object-oriented Programming (ECOOP'93)*, volume 707 of *Lecture Notes in Computer Science*, Kaiserstautern, Germany, July 1993.

[56] *Proceedings of the European Conference on Object-oriented Programming (ECOOP'99)*, volume 1628 of *Lecture Notes in Computer Science*, Lisbon, Portugal, June 1999.

[57] M. Emami, R. Ghiya, and L.J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation (PLDI'94)*, pages 242–256. ACM SIGPLAN Notices, 29(6), June 1994.

[58] D.R. Engler, W.C. Hsieh, and M.F. Kaashoek. 'C: A language for high-level, efficient, and machine-independent dynamic code generation. In POPL'96 [132], pages 131–144.

[59] D.R. Engler and M.F. Kaashoek. DPF: Fast, flexible message demultiplexing using dynamic code generation. In *SIGCOMM Symposium on Communications Architectures and Protocols*, pages 26–30, Stanford University, CA, August 1996. ACM Press.

[60] Java Grande Forum. The Java Grande Forum benchmark suite, 1999. Accessible from `http://www.javagrande.org` and `http://www.epcc.ed.ac.uk/javagrande`.

[61] N. Fujinami. Automatic run-time code generation in C++. In Y. Ishikawa, R.R. Oldehoeft, J.V.W. Reynders, and M. Tholsburn, editors, *Proceedings of the First International Conference on Scientific Computing in Object-Oriented Parallel Environments (ISCOPE'97)*, volume 1343 of *Lecture Notes in Computer Science*, pages 9–16, Marina del Rey, CA, USA, December 1997. Springer-Verlag.

[62] N. Fujinami. Determination of dynamic method dispatches using run-time code generation. In X. Leroy and A. Ohori, editors, *Proceedings of the Second International Workshop on Types in Compilation (TIC'98)*, volume 1473 of *Lecture Notes in Computer Science*, pages 253–271, Kyoto, Japan, March 1998.

[63] Y. Futamura. Partial evaluation of computation process — an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.

[64] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[65] A.J. Goldberg and D. Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, 1983.

[66] C.K. Gomard and N.D. Jones. A partial evaluator for the untyped lambda calculus. *Journal of Functional Programming*, 1(1):21–69, January 1991.

[67] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[68] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison-Wesley, second edition, 2000.

[69] D. Grove, J. Dean, C. Garrett, and C. Chambers. Profile-guided receiver class prediction. In *OOPSLA'95 Conference Proceedings*, pages 108–123, Austin, TX, USA, October 1995. ACM Press.

[70] B. Guenter, T.B. Knoblock, and E. Ruf. Specializing shaders. In *Computer Graphics Proceedings*, Annual Conference Series, pages 343–350. ACM Press, 1995.

[71] M. Gupta, J. Choi, and M. Hind. Optimizing Java programs in the presence of exceptions. In ECOOP'00 [54].

[72] A. Haraldsson. *A Program Manipulation System Based on Partial Evaluation*. PhD thesis, Linköping University, Sweden, 1977. Linköping Studies in Science and Technology Dissertations N^o 14.

[73] A. Haraldsson. A partial evaluator, its use for compiling iterative statements in Lisp. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 195–202, Tucson, Arizona, January 1978. ACM Press.

[74] G. Hedin. Language support for design patterns using attribute extension. In J. Bosch and S. Mitchell, editors, *ECOOP'97 Workshop Reader*, volume 1357 of *Lecture Notes in Computer Science*, pages 137–140. Springer-Verlag, June 1998.

[75] M. Hind, M. Burke, P. Carini, and J. Choi. Interprocedural pointer alias analysis. *ACM Transactions on Programming Languages and Systems*, 21(4):848–894, July 1999.

[76] U. Hölzle and D. Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 326–336, New York, NY, USA, June 1994. ACM Press.

[77] L. Hornof. *Static Analyses for the Effective Specialization of Realistic Applications.* PhD thesis, University of Rennes I, June 1997.

[78] L. Hornof and J. Noyé. Accurate binding-time analysis for imperative languages: Flow, context, and return sensitivity. *Theoretical Computer Science (TCS)*, 248(1–2), 2000.

[79] L. Hornof, J. Noyé, and C. Consel. Effective specialization of realistic programs via use sensitivity. In P. Van Hentenryck, editor, *Proceedings of the Fourth International Symposium on Static Analysis (SAS'97)*, volume 1302 of *Lecture Notes in Computer Science*, pages 293–314, Paris, France, September 1997. Springer-Verlag.

[80] IBM. IBM JDK 1.3, 2000. Accessible from `http://www.ibm.com/java/jdk`.

[81] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In Meissner [109], pages 132–146.

[82] T. Johnsson. Lambda lifting: Transforming programs to recursive equations. In J. Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, number 201 in Lecture Notes in Computer Science, pages 190–203, Nancy, France, September 1985. Springer-Verlag.

[83] N.D. Jones. What *not* to do when writing an interpreter for specialisation. In Danvy et al. [46], pages 216–237.

[84] N.D. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* International Series in Computer Science. Prentice-Hall, June 1993.

[85] N.D. Jones, P. Sestoft, and H. Søndergaard. Mix: a self-applicable partial evaluator for experiments in compiler generation. *Lisp and Symbolic Computation*, 2(1):9–50, 1989.

[86] N.D. Jones, P. Sestoft, and H. Søndergaard. An experiment in partial evaluation: the generation of a compiler generator. In J. Jouannaud, editor, *Rewriting Techniques and Applications*, volume 202 of *Lecture Notes in Computer Science*, pages 124–140. Springer-Verlag, 1985.

[87] N.D. Jones, P. Sestoft, and H. Søndergaard. Mix: a self-applicable partial evaluator for experiments in compiler generation. *Lisp and Symbolic Computation*, 2(1):9–50, 1989.

[88] S.C. Khoo and R.S. Sundaresh. Compiling inheritance using partial evaluation. In *Partial Evaluation and Semantics-Based Program Manipulation (PEPM'91)*, pages 211–222, New Haven, CT, USA, September 1991. ACM SIGPLAN Notices, 26(9).

[89] G. Kiczales, 1999. Personal communication with author at ECOOP'99, Lisbon, Portugal.

[90] G. Kiczales, 2000. Personal communication with author at OCM'2000, Nantes, France.

[91] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Aksit and S. Matsuoka, editors, *Proceedings of the European Conference on Object-oriented Programming (ECOOP'97)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242, Jyväskylä, Finland, June 1997. Springer.

[92] S.C. Kleene. *Introduction to Metamathematics*. Van Nostrand, 1952.

[93] P. Kleinrubatscher, A. Kriegshaber, R. Zöchling, and R. Glück. Fortran program specialization. In U. Meyer and G. Snelting, editors, *Workshop Semantikgestützte Analyse, Entwicklung und Generierung von Programmen*, pages 45–54. Justus-Liebig-Universität, Giessen, Germany, 1994. Report No. 9402.

[94] T.B. Knoblock and E. Ruf. Data specialization. In PLDI'96 [129], pages 215–225. Also TR MSR-TR-96-04, Microsoft Research, February 1996.

[95] S. Krishnamurthi, Y. Erlich, and M. Felleisen. Expressing structural properties as language constructs. In S.D. Swierstra, editor, *Programming Languages and Systems, 8th European Symposium on Programming (ESOP'99)*, volume 1576 of *Lecture Notes in Computer Science*, pages 258–272. Springer-Verlag, 1999.

[96] J.L. Lawall. Faster Fourier transforms via automatic program specialization. In J. Hatcliff, T.Æ. Mogensen, and P. Thiemann, editors, *Partial Evaluation—Practice and Theory. Proceedings of the 1998 DIKU International Summerschool*, volume 1706 of *Lecture Notes in Computer Science*, pages 338–355, Copenhagen, Denmark, 1999. Springer-Verlag.

[97] J.L. Lawall and G. Muller. Efficient incremental checkpointing of Java programs. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 61–70, New York, NY, USA, June 2000. IEEE.

[98] P. Lee and M. Leone. Optimizing ML with run-time code generation. In PLDI'96 [129], pages 137–148.

[99] J.W. Lloyd and J.C. Shepherdson. Partial evaluation in logic programming. *Journal of Logic Programming*, 11:217–242, 1991.

[100] B.N. Locanthi. Fast bitblt() with asm() and cpp. In *European UNIX Systems User Group Conference Proceedings*, pages 243–259, AT&T Bell Laboratories, Murray Hill, September 1987. EUUG.

[101] O.L. Madsen, editor. *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'92)*, volume 615 of *Lecture Notes in Computer Science*, Utrecht, The Netherlands, 1992. Springer-Verlag.

[102] O.L. Madsen, B. Møller-Pedersen, and K. Nygaard. *Object-oriented programming in the Beta programming language.* Addison-Wesley, Reading, MA, USA, 1993.

[103] H. Makholm. On Jones-optimal specialization for strongly typed languages. In W. Taha, editor, *Semantics, Applications and Implementation of Program Generation*, volume 1924 of *Lecture Notes in Computer Science*, pages 129–148, Montreal, Canada, September 2000. Springer-Verlag.

[104] S. Malabarba, R. Pandey, J. Gragg, E. Bar, and J.F. Barne. Runtime support for type-safe dynamic Java classes. In ECOOP'00 [54].

[105] K. Malmkjær, N. Heintze, and O. Danvy. ML partial evaluation using set-based analysis. In *1994 ACM SIGPLAN Workshop on ML and Its Applications*, pages 112–119, Orlando, Florida, 1994. Technical Report 2265, INRIA Rocquencourt, France.

[106] R. Marlet, S. Thibault, and C. Consel. Mapping software architectures to efficient implementations via partial evaluation. In *Conference on Automated Software Engineering*, pages 183–192, Lake Tahoe, NV, USA, November 1997. IEEE Computer Society.

[107] M. Marquard and B. Steensgaard. Partial evaluation of an object-oriented imperative language. Master's thesis, University of Copenhagen, April 1992.

[108] T.D. Meijler, S. Demeyer, and R. Engel. Making design patterns explicit in FACE, a framework adaptive composition environment. In M. Jazayeri and H. Schauer, editors, *Proceedings of ESEC/FSE'97*, volume 1301 of *Lecture Notes in Computer Science*, pages 94–110. Springer-Verlag, September 1997.

[109] L. Meissner, editor. *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'99)*, volume 34(10) of *ACM SIGPLAN Notices*, Denver, Colorado, USA, November 1999. ACM Press.

[110] A. Le Meur and C. Consel. Generic software component configuration via partial evaluation, August 2000. Presented at the Product Line Architecture Workshop at The First Software Product Line Conference in Denver, Colorado.

[111] B. Meyer. *Eiffel: The language.* Object-Oriented Series. Prentice Hall, New York, NY, 1992.

[112] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised).* The MIT Press, 1997.

[113] T. Mogensen. Binding-time analysis for polymorphically typed higher order languages. In J. Diaz and F. Orejas, editors, *TAPSOFT'89,*

Barcelona, Spain, volume 352 of *Lecture Notes in Computer Science*, pages 298–312. Springer-Verlag, March 1989.

[114] T. Mogensen. Constructor specialization. In D. Schmidt, editor, *ACM SIGPLAN Symposium on Partial Evaluation and Semantics Based Program Manipulation*, pages 22–32, 1993.

[115] B. Moura. *Bridging the Gap between Functional and Imperative Languages.* PhD thesis, University of Rennes I, April 1997.

[116] G. Muller, R. Marlet, E.N. Volanschi, C. Consel, C. Pu, and A. Goel. Fast, optimized Sun RPC using automatic program specialization. In *Proceedings of the 18th International Conference on Distributed Computing Systems*, pages 240–249, Amsterdam, The Netherlands, May 1998. IEEE Computer Society Press.

[117] G. Muller and U.P. Schultz. Harissa: A hybrid approach to Java execution. *IEEE Software*, pages 44–51, March 1999.

[118] P. Naur. Revised report on the algorithmic language ALGOL 60. *Communications of the ACM*, 6:1–17, 1963.

[119] H.R. Nielson and F. Nielson. Automatic binding-time analysis for a typed lambda-calculus. *Science of Computer Programming*, 10(2):139–176, April 1988.

[120] F. Noël, L. Hornof, C. Consel, and J. Lawall. Automatic, template-based run-time specialization: Implementation and experimental study. In *International Conference on Computer Languages*, pages 132–142, Chicago, IL, May 1998. IEEE Computer Society Press. Also available as IRISA report PI-1065.

[121] K. Nygaard and O.J. Dahl. Simula 67. In R.W. Wexelblat, editor, *History of Programming Languages.* ACM Press, 1986.

[122] *OOPSLA'96 Conference Proceedings*, volume 31, 10 of *ACM SIGPLAN Notices*, New York, NY, USA, October 1996. ACM Press.

[123] *OOPSLA'97 Conference Proceedings*, Atlanta, GA, USA, October 1997. ACM Press.

[124] N. Oxhøj, J. Palsberg, and M. Schwartzbach. Making type inference practical. In Madsen [101], pages 329–349.

[125] J. Palsberg and M. Schwartzbach. Object-oriented type inference. In N. Meyrowitz, editor, *OOPSLA'91 Conference Proceedings*, volume 26(11), pages 146–161. ACM Press, November 1991.

[126] *Partial Evaluation and Semantics-Based Program Manipulation (PEPM'93)*, Copenhagen, Denmark, June 1993. ACM Press.

[127] *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'97)*, Amsterdam, The Netherlands, June 1997. ACM Press.

[128] *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation (PLDI'95)*. ACM SIGPLAN Notices, 30(6), June 1995.

[129] *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation (PLDI'96)*, Philadelphia, PA, USA, May 1996. ACM SIGPLAN Notices, 31(5).

[130] J. Plevyak and A.A. Chien. Precise concrete type inference for object-oriented languages. In *OOPSLA'94 Conference Proceedings*, volume 29:10 of *SIGPLAN Notices*, pages 324–324. ACM Press, October 1994.

[131] M.P. Plezbert and R.K. Cytron. Does "just in time" = "better late than never"? In *Conference Record of the* 24*th* *Annual ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages (POPL'97)*, pages 120–131, Paris, France, January 1997. ACM Press.

[132] *Conference Record of the* 23*rd* *Annual ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages (POPL'96)*, St. Petersburg Beach, FL, USA, January 1996. ACM Press.

[133] Compose Project. Tempo documentation. Accessible from the Tempo homepage: `http://www.irisa.fr/compose/tempo`.

[134] C. Pu, T. Autrey, A. Black, C. Consel, C. Cowan, J. Inouye, L. Kethana, J. Walpole, and K. Zhang. Optimistic incremental specialization: Streamlining a commercial operating system. In *Proceedings of the 1995 ACM Symposium on Operating Systems Principles (SOSP'95)*, pages 314–324, Copper Mountain Resort, CO, USA, December 1995. ACM Operating Systems Reviews, 29(5), ACM Press.

[135] C. Pu, H. Massalin, and J. Ioannidis. The Synthesis kernel. *Computing Systems*, 1(1):11–32, Winter 1988.

[136] W. Pugh. Compressing Java class files. In *Proceedings of the ACM SIGPLAN'99 Conference on Programming Language Design and Implementation (PLDI'99)*, pages 247–258, Atlanta, G.A., USA, May 1999.

[137] R.K. Raj, E. Tempero, H.M. Levy, A.P. Black, N.C. Hutchinson, and E. Jul. Emerald: A general-purpose programming language. *Software — Practice and Experience*, 21(1):91–118, January 1991.

[138] D. Rayside, E. Mamas, and E. Hons. Compact Java binaries for embedded systems. In *Proceedings of CASCON'99*, pages 1–14, Toronto, Ontario, November 1999.

[139] S.A. Romanenko. Arity raiser and its use in program specialization. In N. Jones, editor, *Third European Symposium on Programming (ESOP'90)*, number 432 in Lecture Notes in Computer Science, pages 340–360, Copenhagen, Denmark, May 1990. Springer-Verlag.

[140] E. Ruf. Context-insensitive alias analysis reconsidered. In PLDI'95 [128], pages 13–22.

[141] J.C. Russ. *The Image Processing Handbook.* CRC Press, Inc., second edition, 1995.

[142] U.P. Schultz. Black-box program specialization. In W. Weck, J. Bosch, and C. Szyperski, editors, *Proceedings of the Fourth International Workshop on Component-Oriented Programming (WCOP'99)*, Lisbon, Portugal, 1999. Technical Report 17/99, Department of Software Engineering and Computer Science, University of Karlskrona/Ronneby.

[143] U.P. Schultz, J. Lawall, C. Consel, and G. Muller. Towards automatic specialization of Java programs. In ECOOP'99 [56], pages 367–390.

[144] U.P. Schultz, J.L. Lawall, C. Consel, and G. Muller. Specialization patterns. In ASE'00 [10], pages 197–206.

[145] M. Shaw and D. Garlan. *Software Architecture.* Prentice Hall, 1996.

[146] O. Shivers. *Control-Flow Analysis of Higher-Order Languages or Taming Lambda.* PhD thesis, Carnegie Mellon University, May 1991. Technical Report CMU-CS-91-145.

[147] B. Stroustrup. *The C++ programming language.* Addison-Wesley, second edition, 1992.

[148] Sun Microsystems, Inc. *JavaCard 2.1 Virtual Machine Specification*, March 1999. Accessible from `http://www.javasoft.com/products/javacard`.

[149] Sun Microsystems, Inc. Sun JDK 1.2.2, 1999. Accessible from `http://java.sun.com/products/j2se`.

[150] Sun Microsystems, Inc. Hotspot, 2000. Accessible from `http://java.sun.com/products/hotspot`.

[151] Sun Microsystems, Inc. Sun JDK 1.3, 2000. Accessible from `http://java.sun.com/products/j2se`.

[152] W. Taha and T. Sheard. Multi-state programming with explicit annotations. In PEPM'97 [127], pages 203–217.

[153] A.S. Tannenbaum. *Structured Computer Organization.* Prentice-Hall, third edition, 1991.

[154] S. Thibault, C. Consel, R. Marlet, G. Muller, and J. Lawall. Static and dynamic program compilation by interpreter specialization. *Higher-Order and Symbolic Computation (HOSC)*, 13(3):161–178, 2000.

[155] F. Tip and P.F. Sweeney. Class hierarchy specialization. In OOPSLA'97 [123], pages 271–285.

[156] T. Tourwe and W. De Meuter. Optimizing object-oriented languages through architectural transformations. In S. Jähnichen, editor, *Compiler Construction - 8th International Conference (CC'99)*, volume 1575 of *Lecture Notes in Computer Science*, pages 244–258, Amsterdam, The Netherlands, March 1999. Springer-Verlag.

[157] D. Ungar and R.B. Smith. Self: The power of simplicity. In *OOPSLA'87 Conference Proceedings*, volume 22, 12 of *ACM SIGPLAN Notices*, pages 227–242, December 1987.

[158] T.L. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, June 1995.

[159] T.L. Veldhuizen. C++ templates as partial evaluation. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'98)*, pages 13–18, San Antonio, TX, USA, January 1999. ACM Press.

[160] T.L. Veldhuizen. Just when you thought your little language was safe: "Expression Templates" in Java. IUCS Technical Report 539, Extreme Computing Laboratory, Indiana University Computer Science Department, July 2000.

[161] E.N. Volanschi. *An Automatic Approach to Specializing System Components*. PhD thesis, University of Rennes I, February 1998.

[162] E.N. Volanschi, C. Consel, G. Muller, and C. Cowan. Declarative specialization of object-oriented programs. In OOPSLA'97 [123], pages 286–300.

[163] P. Wadler. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science (TCS)*, 73:231–248, 1990.

[164] W.E. Weihl. Interprocedural data flow analysis in the presence of pointers, procedure variables and label variables. In *Conference Record of the Seventh annual ACM Symposium on Principles of Programming Languages (POPL'80)*, pages 83–94. ACM Press, January 1980.

[165] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In Meissner [109], pages 187–206.

[166] R.P. Wilson, R.S. French, C.S. Wilson, S.P. Amarasinghe, J.M. Anderson, S.W.K. Tjiang, S. Liao, C. Tseng, M.W. Hall, M.S. Lam, and J.L. Hennessy. SUIF: An infrastructure for research on parallelizing and optimizing compilers. *ACM SIGPLAN Notices*, 29(12):31–37, December 94.

[167] AspectJ 0.7beta6 design notes, 2000. Accessible as `http://aspectj.org/documentation/designNotes`. Xerox Corp.

[168] AspectJ home page, 2000. Accessible as `http://aspectj.org`. Xerox Corp.