Université de Montréal

Simple Optimizing JIT Compilation of Higher-Order Dynamic Programming Languages

par

Baptiste Saleil

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Thèse présentée à la Faculté des études supérieures et postdoctorales
en vue de l'obtention du grade de
Philosophiæ Doctor (Ph.D.)
en informatique

Mai 2019

© Baptiste Saleil, 2019

Résumé

Implémenter efficacement les langages de programmation dynamiques demande beaucoup d'effort de développement. Les compilateurs ne cessent de devenir de plus en plus complexes. Aujourd'hui, ils incluent souvent une phase d'interprétation, plusieurs phases de compilation, plusieurs représentations intermédiaires et des analyses de code. Toutes ces techniques permettent d'implémenter efficacement un langage de programmation dynamique, mais leur mise en œuvre est difficile dans un contexte où les ressources de développement sont limitées. Nous proposons une nouvelle approche et de nouvelles techniques dynamiques permettant de développer des compilateurs performants pour les langages dynamiques avec de relativement bonnes performances et un faible effort de développement.

Nous présentons une approche simple de compilation à la volée qui permet d'implémenter un langage en une seule phase de compilation, sans transformation vers des représentations intermédiaires. Nous expliquons comment le *versionnement de blocs de base*, une technique de compilation existante, peut être étendue, sans effort de développement significatif, pour fonctionner interprocéduralement avec les langages de programmation d'ordre supérieur, permettant d'appliquer des optimisations interprocédurales sur ces langages. Nous expliquons également comment le *versionnement de blocs de base* permet de supprimer certaines opérations utilisées pour implémenter les langages dynamiques et qui impactent les performances comme les vérifications de type. Nous expliquons aussi comment les compilateurs peuvent exploiter les représentations dynamiques des valeurs par *Tagging* et *NaN-boxing* pour optimiser le code généré avec peu d'effort de développement. Nous présentons également notre expérience de développement d'un compilateur à la volée pour le langage de programmation Scheme, pour montrer que ces techniques permettent effectivement de construire un compilateur avec un effort moins important que les compilateurs actuels et qu'elles permettent de générer du code efficace, qui rivalise avec les meilleures implémentations du langage Scheme.

Mots clés: machine virtuelle, compilation à la volée, ordre supérieur, langage dynamique, spécialisation de code, Scheme, inter-procédural

Abstract

Efficiently implementing dynamic programming languages requires a significant development effort. Over the years, compilers have become more complex. Today, they typically include an interpretation phase, several compilation phases, several intermediate representations and code analyses. These techniques allow efficiently implementing these programming languages but are difficult to implement in contexts in which development resources are limited. We propose a new approach and new techniques to build optimizing just-in-time compilers for dynamic languages with relatively good performance and low development effort.

We present a simple just-in-time compilation approach to implement a language with a single compilation phase, without the need to use code transformations to intermediate representations. We explain how *basic block versioning*, an existing compilation technique, can be extended without significant development effort, to work interprocedurally with higher-order programming languages allowing interprocedural optimizations on these languages. We also explain how *basic block versioning* allows removing operations used to implement dynamic languages that degrade performance, such as type checks, and how compilers can use *Tagging* and *NaN-boxing* to optimize the generated code with low development effort. We present our experience of building a JIT compiler using these techniques for the Scheme programming language to show that they indeed allow building compilers with less development effort than other implementations and that they allow generating efficient code that competes with current mature implementations of the Scheme language.

Keywords: VM, JIT, compiler, higher-order, dynamic language, code specialization, Scheme, interprocedural

Contents

Résumé ii

Abstract iii

List of abbreviations xi

List of abbreviations xi

List of tables xii

List of figures xiii

Remerciements 1

Chapter 1. Compiling Dynamic Languages 2

1.1. Background 6

1.1.1. Dynamic Languages 6

1.1.2. Higher-Order Languages 7

1.1.3. JIT Compilation 7

1.1.4. Frameworks 9

1.1.5. Basic Block Versioning 10

1.1.6. Scheme 14

Chapter 2. Compiler Design for Extremely Lazy Specializing Compilation 17

2.1. Introduction 17

2.2. Related Work 18

2.2.1. Tracing 18

2.2.2. Language Implementation Frameworks 19

2.2.3. Generic Intermediate Representations 20

2.3. Building and Executing Abstract Syntax Trees 20

2.4. Source Language Example 22

2.5. Abstract Target Machine 22

2.6. Code Generation from Abstract Syntax Trees 24

2.7. Lazy Compilation from Abstract Syntax Trees 26
2.7.1. Implementation of the Callback Mechanism 26
2.7.2. Implementation of Continuation-Passing Style Code Generation 28

2.8. Code Specialization from Abstract Syntax Trees 30
2.8.1. Implementation of the Contexts and Version Caching 31
2.8.2. Implementation of Code Specialization 33

2.9. Example of Type Test Removal 35

2.10. Optimizations 37
2.10.1. Stack Mapping 38
2.10.1.1. Example 38
2.10.2. Register Allocation 40
2.10.2.1. Example 41
2.10.3. Constant Propagation 42
2.10.3.1. Example 42
2.10.4. Value Unboxing 44
2.10.4.1. Example 44

2.11. Summary 45

Chapter 3. Interprocedural Code Specialization in the Presence of Higher-Order Functions 47

3.1. Introduction 47

3.1.1. Function Calls ... 47
3.1.2. Captured Properties ... 49
3.1.3. Function Returns ... 49
3.2. Related Work ... 50
3.2.1. Function Entry Points ... 50
3.2.1.1. Inline Expansion ... 50
3.2.1.2. Static Analyses ... 51
3.2.1.3. Code Customization ... 51
3.2.1.4. Basic Block Versioning ... 52
3.2.1.5. Tracing ... 53
3.2.2. Call Continuations ... 53
3.2.2.1. Continuation-Passing Style ... 53
3.2.2.2. Basic Block Versioning ... 54
3.3. Flat Closure Representation ... 54
3.3.1. Example ... 55
3.4. Flat Closure Representation Extension ... 56
3.4.1. Example ... 58
3.5. Dynamic Dispatch ... 58
3.5.1. Global layout ... 58
3.5.2. Implementation ... 59
3.5.2.1. Retrieving the Contexts Used at Call Sites ... 61
3.5.3. Optimizations ... 63
3.5.3.1. Slot Preallocation ... 63
3.5.3.2. Generic Entry Point ... 63
3.5.3.3. Specific Entry Points ... 64
3.5.3.4. Heuristics ... 64
3.6. Captured Properties ... 66

3.6.1. Entry Point Table Specialization 67
3.6.2. Example 67
3.6.3. Optimizations 68
3.6.3.1. Limiting the Number of Specialized Tables 68
3.6.3.2. Closure Patching 71
3.7. Continuations 71
3.7.1. Continuation Representation Extension 72
3.7.2. Dynamic Dispatch 72
3.7.3. Multiple Return Values 72
3.7.4. Captured Properties 73
3.7.5. Optimizations 74
3.7.5.1. Fixed Table Layout 74
3.7.6. Example 75
3.8. Compiler Optimizations 75
3.9. Summary 76
Chapter 4. Eager Unboxing Based on Basic Block Versioning 77
4.1. Introduction 77
4.2. Related Work 79
4.2.1. Run Time Value Representation 79
4.2.2. Boxing and Unboxing Operation Removal 82
4.3. Unboxing Based on Basic Block Versioning 83
4.3.1. Example 84
4.4. Eager Unboxing 85
4.5. Comparison of Tagging and NaN-boxing 86
4.5.1. Tagging 86
4.5.1.1. Fixnums 88

4.5.1.2. Special Values ... 89
4.5.1.3. Flonums ... 89
4.5.1.4. Memory Allocated Objects ... 89
4.5.1.5. Summary ... 91
4.5.2. NaN-boxing ... 91
4.5.2.1. Fixnums ... 94
4.5.2.2. Special Values ... 95
4.5.2.3. Flonums ... 95
4.5.2.4. Memory Allocated Objects ... 96
4.5.2.5. Summary ... 96
4.5.3. Choosing a Representation to Decrease Development Effort ... 97
4.6. Garbage Collector ... 98
4.6.1. Separate Stack ... 98
4.6.2. Garbage Collection Stack Maps ... 99
4.6.3. Conservative Garbage Collection ... 99
4.7. Register Allocation Extension ... 100
4.8. Summary ... 101
Chapter 5. LC ... 102
5.1. Overview ... 102
5.2. Implementation From Scratch ... 103
5.3. Code Transformations and Static Analyses ... 104
5.4. Compilation Contexts ... 105
5.4.1. Type Property ... 107
5.4.2. Register Allocation Property ... 107
5.4.3. Environment ... 107
5.4.4. Other Information ... 108

5.4.5. Context Handling . . . 108
5.4.6. Versioning and Version Caching . . . 109
5.5. Garbage Collector . . . 110
5.6. Gambit Bridge . . . 111
5.6.1. Implementation . . . 112
5.7. Tools . . . 113
5.7.1. Data Extraction . . . 113
5.7.2. Benchmarking . . . 113
5.7.3. Data Visualization . . . 114
5.8. ECOOP Artifact . . . 116
5.9. Summary . . . 116
Chapter 6. Results . . . 117
6.1. Benchmarks . . . 118
6.1.1. Methodology . . . 118
6.1.2. Setup . . . 119
6.2. Intraprocedural Basic Block Versioning and Compilation Design . . . 119
6.2.1. Number of Type Tests . . . 120
6.2.2. Generated Code Size . . . 122
6.2.3. Execution Time . . . 123
6.2.4. Compilation Time . . . 124
6.2.5. Summary . . . 124
6.3. Interprocedural Extensions . . . 125
6.3.1. Number of Type Tests . . . 125
6.3.2. Generated Code Size . . . 127
6.3.3. Function and Continuation Tables Size . . . 128
6.3.4. Execution Time . . . 130

6.3.5. Compilation Time ... 132
6.3.6. Summary ... 134

6.4. Eager Unboxing ... 134
6.4.1. Number of Executed Boxing and Unboxing Operations ... 135
6.4.1.1. Heterogeneous Vectors ... 135
6.4.1.2. Homogeneous Vectors ... 136
6.4.2. Execution Time ... 139
6.4.3. Compilation and Garbage Collection Time ... 141
6.4.4. Summary ... 141

6.5. General Performance ... 142
6.5.1. Ahead-Of-Time Optimizing Compilation ... 142
6.5.2. Just-In-Time Optimizing Compilation ... 144

6.6. Summary ... 146

Chapter 7. Future Work ... 147

7.1. Compound Data Types ... 147

7.2. Properties Used for Specialization ... 150
7.2.1. Constants ... 150
7.2.2. Register Allocation ... 152

7.3. Lazy Unboxing in Other Contexts ... 152

Chapter 8. Conclusion ... 154

Bibliography ... 156

List of abbreviations

AOT	Ahead-Of-Time
AST	Abstract Syntax Tree
BBV	Basic Block Versioning
CFG	Control Flow Graph
CPS	Continuation-Passing Style
CSE	Common Subexpression Elimination
CSV	Comma-Separated Values
GC	Garbage Collector
IR	Intermediate Representation
JIT	Just-In-Time
JVM	Java Virtual Machine
NaN	Not a Number
OSR	On-Stack Replacement
PIC	Polymorphic Inline Cache / Polymorphic Inline Caching
qNaN	Quiet NaN
sNaN	Signaling NaN
TCO	Tail Call Optimization
VM	Virtual Machine

List of tables

3.1 Percentage of function calls executed per number of arguments for each benchmark. 65

3.2 Percentage of tables per number of table specializations. 70

4.1 Summary of the cost associated with boxing and unboxing operations for various value representations (light colored squares are the least expensive operations). . . 80

6.1 Memory used by the function and continuation entry point tables when using interprocedural code specialization. This table also shows the number of contexts used for function entry point specialization and the generated code size. 129

6.2 Number of executed flonum boxing and unboxing operations, using heterogeneous vectors, with eager unboxing relative to naive mode. 135

6.3 Number of executed flonum boxing and unboxing operations, using f64vectors, with eager unboxing relative to naive mode. 136

6.4 Number of executed flonum boxing and unboxing operations, using heterogeneous vectors, with the Gambit Scheme compiler relative to LC in naive mode. 138

List of figures

1.1 Typical Scheme code expansion including type dispatches for a compiler supporting fixnums and flonums. 10

1.2 Typical control flow graph generated by a compiler for the function `abs`. 12

1.3 Compilation state after the first block is generated. 13

1.4 Compilation state after the whole function is generated for a call with a negative fixnum. 14

1.5 Compilation state after the whole function is generated for a call with a negative fixnum and a positive flonum. 15

2.1 Grammar of a simple language, based on S-expressions, used as an example in this chapter. 22

2.2 Abstract stack-based target machine definition. 23

2.3 Typical pattern matching used to generate code for an input S-expression. 25

2.4 Implementation of the callback mechanism on top of the abstract machine. 27

2.5 Typical pattern matching used to generate code for an input S-expression with lazy compilation. 28

2.6 Implementation of the contexts and version caching. 32

2.7 Lazy compilation and code specialization for an input S-expression. 34

2.8 Grammar of a simple language with variables, constants and if. 36

2.9 Example of code expansion to insert type tests for primitives. 36

2.10 Lazy compilation of the expression `(f64vector-ref v 2)` with generated x86_64 code (Intel syntax). 39

2.11 Lazy compilation of the expression (f64vector-ref v 2) using lazy register allocation with generated x86_64 code (Intel syntax). 41
2.12 Lazy compilation of the expression (f64vector-ref v 2) using constant propagation with generated x86_64 code (Intel syntax). 43
2.13 Lazy compilation of the expression (f64vector-ref v 2) using value unboxing with generated x86_64 code (Intel syntax). 45
3.1 Scheme code of an arithmetic sequence generator with a common step of 1. 48
3.2 Flat closure representation. 55
3.3 Flat closures created for the function f of the example in Figure 3.1. 55
3.4 Extended flat closure representation with an entry point table. 57
3.5 Closures using entry point tables created for the function f of the example in Figure 3.1. 57
3.6 Example of x86_64 function call sequence using flat closures and intraprocedural only code specialization (Intel syntax). 60
3.7 Example of x86_64 function call sequence using flat closures and interprocedural code specialization with entry point tables (Intel syntax). 60
3.8 Code of Figure 3.7 extended to pass the index associated with the $\mathcal{P}_{\text{call}}$ used at the call site to callee function stubs. When the call triggers a stub, the compiler retrieves the index from the register r11 and retrieves $\mathcal{P}_{\text{call}}$ from this index. When the call does not trigger a stub, the index value is not used. 62
3.9 Closures using specialized entry point tables created for the function f of the example in Figure 3.1. 68
4.1 Scheme function generating the nth Fibonacci number. 78
4.2 Example of code causing eager unboxing to generate an unnecessary operation. . . 86
4.3 Tagging representation and example of boxed values with n=64 and t=2. 87
4.4 Example of a fixnum addition with no unboxing operations. 88

4.5 Implementation of boxing and unboxing operations for tagged fixnums (using the tag 0) on x86_64 machines (Intel syntax). The fixnum is initially in the rax register. 88

4.6 Implementation of boxing and unboxing operations for tagged flonums on x86_64 machines (Intel syntax). The flonum is initially in the xmm0 floating-point register for the boxing operation and in the rax register for the unboxing operation. The boxing code uses a bump pointer allocator. r8 is the heap pointer and r9 is the heap limit address. 90

4.7 Implementation of the Scheme vector-ref primitive using a single x86_64 instruction. rax is the destination register, rbx is the boxed vector and rcx is the boxed fixnum index. 90

4.8 Example of canonical representations for NaNs. 92

4.9 NaN-boxing representation and example of boxed values. 93

4.10 Implementation of boxing and unboxing operations for NaN-boxed fixnums on x86_64 machines (Intel syntax). The fixnum is initially in the eax register (lower 32 bits of the rax register). The mov instruction in the unboxing code zero-extends the value in eax to rax (the 32 most significant bits are cleared and the 32 least significant bits, representing the fixnum, are unchanged). However, the unboxing operation is typically not needed on x86_64 because the 32-bit instructions can directly use the 32-bit registers (including eax). Note that the cdqe instruction can be used if the value needs to be sign-extended instead of zero-extended. 95

4.11 Implementation of boxing and unboxing operations for NaN-boxed memory allocated objects on x86_64 machines (Intel syntax). The object is initially in the rax register. We assume that the address of the object fits in 48 bits and is positive. For the unboxing operation, the constant is loaded with a mov instruction because the other instructions limit the constants to 32 bits. 96

5.1 Internal representation of a context in LC. The comments show examples of possible values for each field of the context. 106

5.2 Sample of use of the bridge to access the host environment using non-standard Scheme functions available in Gambit. 112

5.3 Example of using the `--stats` option of LC to collect execution information of a Scheme program. 114

5.4 Visualization of the versions generated for a program from a report generated by LC. 115

6.1 Proportion of remaining executed tests using intraprocedural BBV relative to naive compilation for LC with a version limit of 5 (naive compilation is 100%). 120

6.2 Scheme code of the `takl` benchmark. This benchmark implements the Takeuchi function using lists as counters. 121

6.3 Size of the generated code using intraprocedural BBV relative to naive compilation for LC with a version limit of 5 (naive compilation is 1.0). 122

6.4 Execution time of the generated code using intraprocedural BBV relative to naive compilation for LC with a version limit of 5 (naive compilation is 1.0). 123

6.5 Compilation time of the generated code using intraprocedural BBV relative to naive compilation for LC with a version limit of 5 (naive compilation is 1.0). 124

6.6 Proportion of remaining executed tests using interprocedural BBV relative to intraprocedural BBV for LC with a version limit of 5 (intraprocedural BBV is 100%). 126

6.7 Proportion of remaining executed tests using intraprocedural and interprocedural BBV relative to naive compilation for LC with a version limit of 5 (naive compilation is 100%). 126

6.8 Size of the generated code using interprocedural BBV relative to intraprocedural BBV for LC with a version limit of 5 (intraprocedural BBV is 1.0). 127

6.9 Size of the generated code using intraprocedural and interprocedural BBV relative to naive compilation for LC with a version limit of 5 (naive compilation is 1.0). . . 127

6.10 Execution time of the generated code using interprocedural BBV relative to intraprocedural BBV for LC with a version limit of 5 (intraprocedural BBV is 1.0). 130

6.11 Execution time of the generated code using intraprocedural BBV and interprocedural BBV relative to naive compilation for LC with a version limit of 5 (naive compilation is 1.0). 130

6.12 Compilation time of the generated code using interprocedural BBV relative to intraprocedural BBV for LC with a version limit of 5 (intraprocedural BBV is 1.0). 132

6.13 Compilation time of the generated code using intraprocedural and interprocedural BBV relative to naive compilation for LC with a version limit of 5 (naive compilation is 1.0). 132

6.14 Execution time of the generated code of flonum benchmarks using eager unboxing (with normal heterogeneous vectors and f64vectors) relative to naive unboxing (respectively with heterogeneous vectors and f64vectors) for LC with a version limit of 5 (naive unboxing is 1.0). 139

6.15 Execution time of the generated code using NaN-boxing and Tagging with eager unboxing relative to Tagging without eager unboxing for LC with a version limit of 5 (Tagging without eager unboxing is 1.0). 140

6.16 Execution time (excluding GC and compilation time) of the code generated by Gambit and Chez in safe and unsafe modes relative to LC (LC is 1.0). 143

6.17 Execution time (excluding GC time and including JIT compilation time) of the code generated by LC relative to Pycket (Pycket is 1.0). The dark gray bars show the execution time for LC using f64vectors and the light gray bars show the difference using heterogeneous vectors. 145

7.1 Scheme code to compute the proper divisors of a fixnum and the sum of these divisors. 148

7.2 Three approaches to implement versioning while avoiding explosions in the number of versions. The approach on the top part using abrupt degradation illustrates how

a hard limit avoids the explosion. The approach on the left side illustrates how heuristics allow gradually reaching the generic version. The approach on the right side illustrates how using a generic version bounded by a static BBV phase avoids falling back to a completely generic version. 151

Remerciements

Le doctorat est un voyage long et difficile. C'est grâce au soutien de certaines personnes que j'ai été capable d'aller jusqu'au bout, et je voudrais ici les remercier.

Merci à Marc, mon directeur. Merci pour tout ce que tu m'as appris, tant sur la compilation que sur la recherche et ses secrets, merci de m'avoir guidé jusqu'au bout. Merci aux membres de mon jury pour avoir lu ma thèse, pour m'avoir donné de bons conseils pour en améliorer la qualité et pour m'avoir fait l'honneur d'assister à ma soutenance. Merci aux membres et anciens membres du laboratoire LTP; grâce à vous, j'ai appris énormément de choses.

Merci à Cécilia, pour ta présence au quotidien. Sans toi, je ne serais jamais allé jusqu'au bout. Merci à ma famille de m'avoir toujours soutenu pendant ce doctorat et d'avoir accepté mon départ. Merci à mes amis pour tous les bons moments qu'on a passés ensemble, pour m'avoir fait penser à autre chose dans les moments difficiles, merci PouerMouer & Co. Merci Pito.

Chapter 1

Compiling Dynamic Languages

Dynamic languages allow rapid development by decreasing programmer constraints with dynamic features such as dynamic typing and late binding. However, they generally suffer from relatively poor performance compared to static languages. Dynamic languages are increasingly used to build complex applications including desktop and web applications. For example, Python is increasingly used for scientific computation and JavaScript, originally used as a scripting language for the web now allows building complete desktop, client side and server side web applications. Initial implementations of dynamic languages are often interpreted but because it is now important that programs written in dynamic languages execute efficiently, interpretation falls short of expectations.

Ahead-Of-Time (AOT) compilation is a technique widely used for static languages such as *C*, *C++*, *Go* and *Rust*. An AOT compiler compiles the whole program to the target language, which in efficient systems is generally binary code the target processor can execute. Once the program is compiled to the target language, it is distributed to users. AOT compilers can use several compilation passes, analyses and optimizations to increase performance of the generated code at the cost of slower compilation. They generally use one or several intermediate languages, referred to as Intermediate Representation (IR) in this thesis, each more suitable for specific analyses and optimizations. AOT compilation is also used with dynamic languages. For example, most compilers for the Scheme programming language [**134**] use AOT compilation to compile Scheme programs to C or machine code.

However, AOT compilation is not the best fit for dynamic languages because the compiler cannot effectively predict the dynamic properties of the compiled program at compile time. For example, in a language featuring dynamic typing, it is possible a variable is bound to

values of different types at execution time. An AOT compiler may choose to be conservative and associate the union of those types to the variable even though only a single type is used in normal situations, thus losing optimization opportunities.

Another approach is to use Just-In-Time (JIT) compilation. In JIT compilation, the source program is distributed to the users and a Virtual Machine (VM) that includes a JIT compiler executes it. When it is executed, parts of the program are compiled on demand or when the VM determines that they are worth compiling (typically if they are frequently executed).

Because compilation occurs when the program is executed, the compiler has more opportunities to generate efficient code using optimizations specialized for the current execution of the program. However JIT compilation introduces new issues. In particular, because compilation occurs when the program is executed by the VM, the time to compile the program is part of the total execution time and consequently the compiler must limit the use of expensive analyses, transformations and optimizations.

Today, production implementations of dynamic languages use VMs with JIT compilers. A typical example is JavaScript. Because JavaScript's popularity is high and it is now used to build complex web applications, web browsers integrate high performance JIT compilers (V8 with Google Chrome [**5**], JavaScriptCore with Safari [**4**], Chakra with Edge [**1**], SpiderMonkey with Firefox [**3**]). To achieve high performance, VMs have been highly complexified over the years. They generally include multiple JIT compilers (multi-tier VMs), involving recompilations, deoptimizations and using complex techniques such as On-Stack Replacement (OSR) [**68**]. For example, the JavaScriptCore VM uses 6 execution engines for JavaScript including 5 JIT compilers and an interpreter [**106**].

Each compiler of a multi-tier VM offers a different trade-off between the time it takes to compile the code and the efficiency of the generated code. A typical approach is to start executing the program using an interpreter. When a hot spot (i.e. a piece of code that is *often* executed) is detected, the next level is used, generally involving a baseline JIT compiler compiling this piece of code with a limited number of optimizations to limit the compilation time. When the VM detects that this piece of code is even more often executed, the next level is triggered, and the code is recompiled to generate even more efficient code at the cost of increasing the time spent to compile it. Choosing between the optimization levels can also

be directed by code profiling. Code profiling allows the VM to infer the dynamic properties of the program using some kind of monitoring (for example, static analysis and type feedback).

Building high performance VMs for dynamic languages is a hard task that requires considerable resources (including funding, time and workforce) [**25**]. Limited resources are the norm when prototyping new programming languages, implementing domain specific languages and in academic research. To decrease the effort required to build efficient implementations, researchers instead focus on implementing dynamic languages using frameworks specially designed for this purpose [**144, 141, 109, 24, 117, 79**]. These frameworks allow significantly decreasing development effort. Unfortunately, the language implementation will have limited control over the underlying implementation of its execution environment. In addition, using such frameworks adds an important software dependence to the implementation and puts at risk its lifetime.

In the last decades, new compilation techniques such as Tracing [**56**], Specialization by Need [**108**] and Basic Block Versioning (BBV) [**37**] have been designed. They allow dynamic language implementations to achieve relatively good performance while significantly decreasing development effort. One of these techniques, Basic Block Versioning, allows a JIT compiler to generate good quality code quickly using code specialization, without requiring previous interpretation or recompilations. Moreover, no static analysis is required to infer the dynamic properties of the program. Relevant properties are instead lazily inferred by BBV, decreasing the complexity of the system and the compilation time.

Our thesis is that VMs for higher-order dynamic languages can be built on top of BBV, with appropriate extensions, to achieve competitive performance and using a single JIT compilation level with relatively low development effort. Our work shows how classical optimizations used for dynamic languages can be applied and extended in this context.

Contributions.

Our work makes four contributions:

- An extension to BBV allowing JIT compilers to lazily optimize code interprocedurally.
- An experimental study of an implementation of BBV for Scheme, a higher-order dynamic programming language.
- A new compiler design for optimizing JIT compilers offering a good trade-off between performance and simplicity / low development effort.
- A study of value representation in JIT compilers in the context of Scheme, with a focus on development effort and how wisely choosing a representation while using BBV allows optimizing code.

Structure of this thesis.

The rest of the thesis is structured as follow.

Chapter 2 presents our compiler design. We explain how BBV can be implemented in an optimizing JIT compiler to generate efficient code but with simplicity and low development effort in mind. We also explain how classical optimizations can easily be applied with this design.

Chapter 3 presents our BBV extension allowing interprocedural specialization of code in the presence of higher-order functions. It extends classical closure and call continuation representations.

Chapter 4 presents the two most used value representations in JIT compilers for dynamic languages: Tagging and NaN-boxing. We explain how the optimization opportunities of BBV can be exploited with these representations to generate efficient code without significantly increasing the complexity of the system and development effort.

Chapter 5 presents LC [**110**], our JIT compiler for the Scheme programming language we have built to evaluate the techniques presented in this thesis.

Chapter 6 presents the results of our experiments with LC. We present the impact of the techniques presented in other chapters and compare our approach to existing mature Scheme systems.

In Chapters 7 and 8, we present future work and conclude.

1.1. Background

1.1.1. Dynamic Languages

A dynamic language is a language offering a set of dynamic features to the programmer. Examples of dynamic features are dynamic typing, late binding and dynamic code loading. The LISP programming language, invented by John McCarthy in 1960 [**89**] is considered as the first dynamic language and has pioneered several dynamic features including dynamic typing and garbage collection [**77**]. Dynamic features have an effect happening when the program is executed that can be difficult or impossible to predict when compiling the program. Consequently, to implement these features, the work must be done at execution instead of at compilation. For example, the use of boxes and boxing and unboxing operations as well as dynamic type tests are used in a languages offering dynamic typing to ensure the safety of the language [**63, 131**]. In a similar way, dispatches are added in the generated code to implement late binding in dynamic languages. Because the work to implement these features is done at execution, these features impact the performance of languages thus dynamic languages are typically considered slower than static languages. Previous work aims to efficiently implement dynamic languages by decreasing the impact of these operations, including for the Scheme language [**65, 125, 142**].

In the 80's, research has been done on the implementation of the Smalltalk programming language [**62**], invented by Alan Kay in 1972. This research significantly improved the performance of dynamic languages and pioneered classical concepts and optimizations of JIT compilation such as on-demand compilation, code caching and inline caching [**44**].

Research on the implementation of the Self programming language, created by David Ungar and Randall Smith in 1987 [**137**], also pioneered several optimizations of JIT compilation of dynamic languages that are still used in current implementations of dynamic languages. This is for example the case for Polymorphic Inline Caches [**66**], Code Customization [**32**], Type Feedback [**69**] and Maps [**33**].

More recently, because of its increasing popularity to build web applications, research for dynamic languages is mainly done for the implementation of the JavaScript language. New compilation techniques have thus been discovered after the 00's. This is for example the case for Tracing (presented in more details in Section 2.2.1), that was applied for the first time to

a dynamic language in TraceMonkey [**56**], the Mozilla's implementation of the JavaScript language. This is also the case for Basic Block Versioning [**37**] presented in detail in Section 1.1.5.

1.1.2. Higher-Order Languages

A higher-order language is a language offering higher-order functions. A higher-order function is a function taking one or more functions as arguments or returning a function. Examples of higher-order languages are Scheme, Clojure, JavaScript and Ruby. A consequence of using higher-order functions is that several functions can be called at a given call site and that it can be impossible to predict the set of functions that can be called at a given call site. Dynamic properties of programs are then more difficult to predict.

The use of lexical scoping and higher-order functions requires using lexical closures [**84**] to represent a function with captured variables. Higher-order functions and closures usually negatively impact the performance of these languages because function calls require additional work, closures require memory allocations and interprocedural optimizations are more difficult to apply. Static analyses such as [**14, 73**] enable interprocedural optimizations in the presence of higher-order functions. *Lambda lifting* [**76**] is a code transformation allowing to decrease the cost of closure memory allocation by transforming free variables into function arguments. Other works such as [**122**] and [**81**] explore new closure representations and closure allocation strategies to decrease the allocation cost.

In this thesis we enable the use of interprocedural optimizations in the presence of higher-order functions without using static analysis. We instead rely on a lazy approach using JIT compilation and extending existing closure representations.

1.1.3. JIT Compilation

There are several ways to implement a programming language. Interpretation allows to quickly build an implementation, but typically suffers from poor performance [**25, 88**].

With AOT compilation, the whole program is compiled before being executed. Because the dynamic properties of programs are more difficult to predict with higher-order dynamic languages, AOT compilation is not adapted to these languages. AOT compilation is thus typically used to implement static languages because, for these languages, the compiler has

extended knowledge on the properties of the compiled program. It is for example used to implement low-level static languages such as C, C++, Rust and Go.

JIT compilation is more popular to implement dynamic languages. The main reason is that using JIT compilation, the program is compiled on demand, meaning that compilation occurs when the program is executed. The compiler can thus use adaptive optimizations [**71, 12**] to generate code that is specialized to the current execution of the program thus specialized to the current dynamic properties of the program.

However, with JIT compilation, new challenges have emerged for language implementers. In particular, because the program is compiled when it is executed, the time the compiler takes to compile the program is included in the total execution time. Research has been done to find new solutions to classical compilation problems with constraints on compilation time. This is for example the case for register allocation [**107, 136, 96, 140, 139**], and static analyses [**64, 87**]. In our implementation, we use a lazy greedy register allocator that does not require analysis and is applied in a single JIT compilation pass.

In a JIT compilation context, there is thus a trade-off between the time the compiler takes to generate the code and the quality of the generated code. To have more control on this trade-off, high-performance execution engines now use multi-tier architectures. In a multi-tier system, the program is executed through several execution levels. Typically, a function is first interpreted when executed. Using interpretation, the execution engine takes no time to generate code but execution is slower. The engine then monitors the execution and can use heuristics to determine that it should be more interesting to compile a function to improve performance, even if some time is spent to compile it. A classical heuristic is to count the number of times a function is executed before deciding to compile it. In advanced multi-tier systems, several JIT compilers can even be successively used to recompile functions with a compiler generating more efficient code than the previous but taking more time to compile it. Examples of multi-tier systems are JavaScriptCore [**106**] that includes an interpreter and several JIT compilers to execute JavaScript, SPUR [**20**] that uses 3 JIT compilers and HHVM [**99**] that uses an interpreter and a JIT compiler.

The use of adaptive optimizations or multi-tier architecture requires recompilations. When a function is recompiled during execution of the program, the compiler must use a mechanism to change the system state to switch execution to the new implementation of the function.

To solve this issue, execution engines use On-Stack Replacement (OSR) [**68, 67, 146**], a technique that is known to be complex. OSR consists of patching the frames in the execution stack to change the system state after a function is recompiled. This technique is for example extensively used in JavaScriptCore [**106**].

Multi-tier systems then became more and more complex over the years and it is now difficult to build an execution engine from scratch offering good performance for a dynamic language. In this thesis, we try to show how far we can get in terms of performance with a system based on a single-tier architecture to decrease the development effort.

A survey of JIT compilation, including JIT compilation of dynamic and static languages is presented in [**15**].

1.1.4. Frameworks

Frameworks have been designed to help building language implementations with relatively low development effort. Because these frameworks include several passes and optimizations that language implementers can rely on, they allow building efficient implementations. However, using these frameworks, language implementers have low control over the low-level implementation. In addition, using a framework generally adds a strong software dependency to the implementation.

Some frameworks allow using an intermediate representation as a target language. This is for example the case for LLVM [**85**]. As an example, OSR has been added to LLVM [**83, 43**] allowing language implementers to exploit this complex technique, for languages targeting LLVM, without having to implement it. The Java Virtual Machine (JVM) [**86**] and the Common Language Runtime [**92**] are other examples of intermediate representations that can be used as targets to build language implementations.

More recently, new frameworks allow generating language implementations. Truffle [**145, 144**] is based on partial evaluation and has been used to build dynamic languages such as Python [**138**] JavaScript [**70**] and Ruby [**118**]. The RPython framework [**109, 24**], based on *meta-tracing*, has also been used to build dynamic languages, including Python [**109, 24**] and Racket [**18, 19**]. Marr et al. compare these two frameworks in [**88**]. The Eclipse OMR framework [**59**] has recently been presented and is used to build implementations for Ruby [**58**], Python and Smalltalk [**132**].

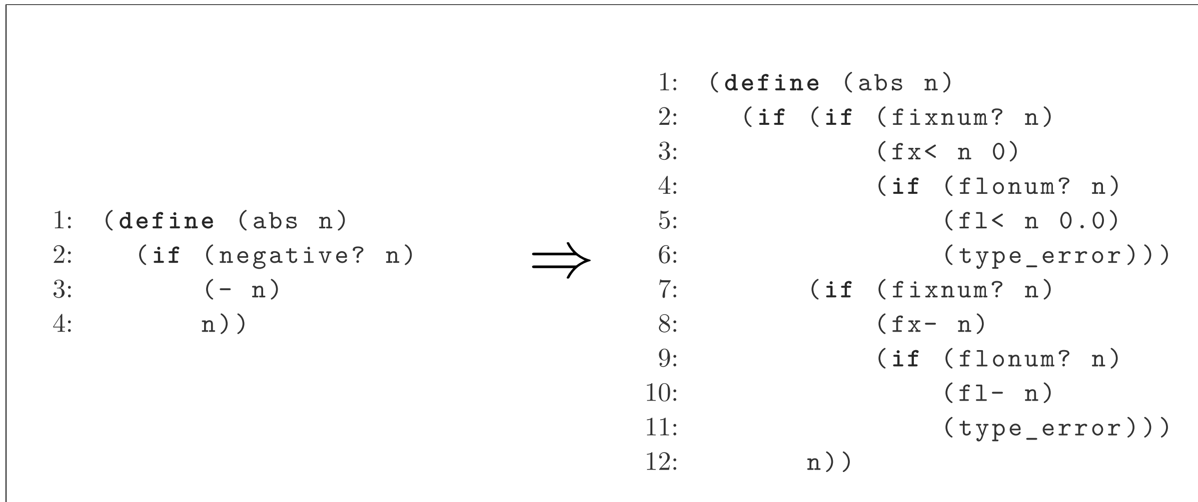


Fig. 1.1. Typical Scheme code expansion including type dispatches for a compiler supporting fixnums and flonums.

Nanopass [**117, 79**] is a domain specific language supporting compiler development. It is based on the use of several passes and intermediate representations helping decreasing the development effort. It is however specialized to AOT compilation.

Other frameworks that were designed to build language implementations are no longer updated [**60, 49**]. Some of them are presented in [**111**].

The work presented in this thesis avoids adding a strong software dependency by using a JIT compilation design allowing to build dynamic language implementations from scratch.

1.1.5. Basic Block Versioning

BBV is a compilation technique using extremely lazy compilation and code specialization. It is used to build compilers for JavaScript [**37**], Scheme [**114**] and Python [**101**].

Lazy compilation allows compilers to discover the dynamic properties of the executed program. These observed properties are taken advantage of by duplicating code to specialize it to the observed properties. This decreases the work done at execution. A typical example is type specialization. In dynamic languages, the primitives[1] may accept operands of different types. For example, the `+` primitive may accept integers, floating point numbers and strings. To ensure the safety of the language, the compiler generates instructions to dispatch the

[1]The primitives of a programming language are the most basic features that are generally used as building blocks of standard libraries provided by the implementations.

execution to the variant of the operator accepting operands of the specific types used at run time. The left side of Figure 1.1 shows a simple Scheme function `abs` computing the absolute value of its argument `n`. This function uses the two primitives `negative?` and unary `-`. When compiling this function, the compiler may not know the types `n` will take at run time. It is possible that the `abs` function is called with `n` being a *fixnum* (Scheme small integer) and later with `n` being a *flonum* (Scheme floating point number). The compiler inserts type tests to dispatch the execution. The right side of Figure 1.1 shows the typical Scheme code used by compilers for the `abs` function after expansion (i.e. after making the type dispatches explicit). To simplify the example, we assume that the compiler supports fixnums and flonums only. Depending on the type of `n`, execution flows to the `fx<` or `fl<` operator and to the `fx-` or `fl-` operator.

Figure 1.2 shows a typical Control Flow Graph (CFG) [**82**] generated by a compiler for the function `abs` after expansion. A CFG is an alternate representation of programs using graphs. Each node of the graph is a *basic block*, a sequence of instructions with branching instructions only at the end of the block. By naively inserting type dispatch tests for each primitive, the type dispatch for the `-` primitive is redundant and this degrades performance. These redundant operations test the type of variables already tested in predecessor blocks. We can see in the CFG that for the `abs` function, type tests in blocks 7 and 8 for the `-` operation are not needed because the type of `n` has already been tested in blocks 1 and 2. The *true* branch of block 4 could directly go to block 11 and the *true* branch of block 5 could directly go to block 10.

BBV removes these redundant tests without analyzing the code nor the CFG. A compiler using BBV compiles at basic block level as opposed to the coarser method level for method-based JIT or loop level for Tracing JIT compilers. Each conditional exit branch of a basic block is an opportunity to discover new dynamic properties of the code. When a basic block is compiled, compilation stubs[2] are created for each exit branch. Information related to the compiler's state is attached to each compilation stub. When a stub is triggered, the compiler

[2]A stub usually is a piece of code that is used as a temporary substitute for other code. In JIT compilation, they are used to delay compilation. A piece of code is inserted in the generated code as a placeholder. When the stub is executed, it calls the compiler to resume compilation. Functions are typically initially represented by compilation stubs, with no other code generated. When a function is called for the first time, its stub is executed and the code at the entry point of the function is generated for subsequent calls.

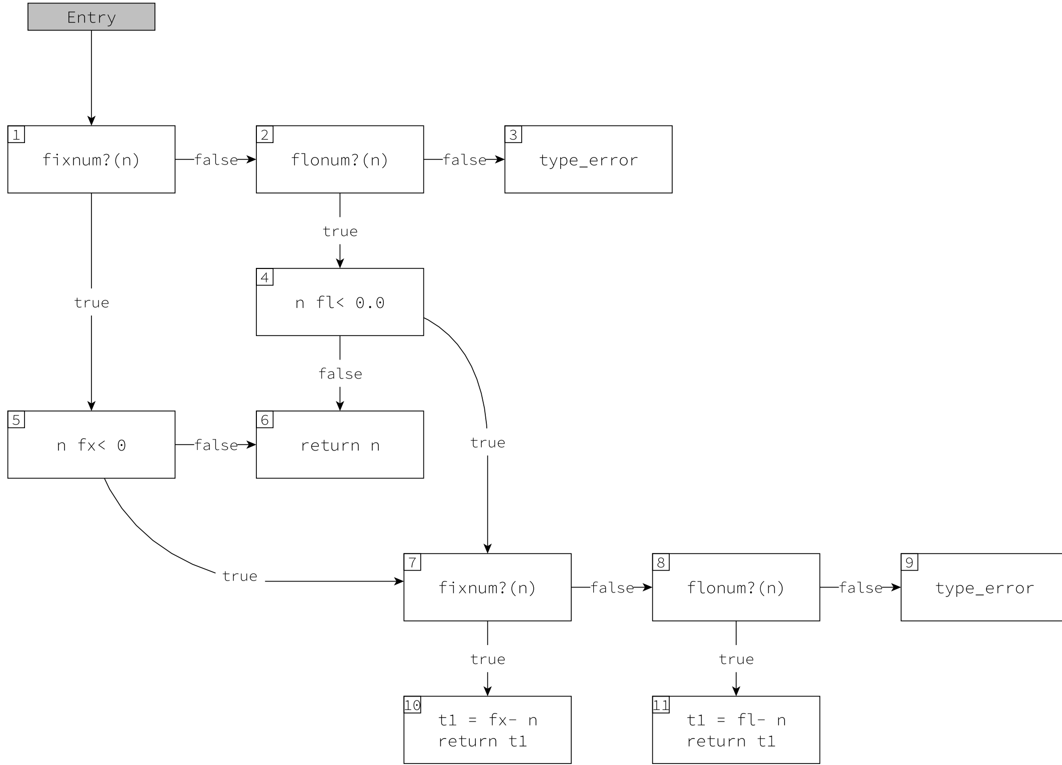


Fig. 1.2. Typical control flow graph generated by a compiler for the function `abs`.

is called to generate the successor block, at the branch destination, specialized using this information. In our example, when the `abs` function is called for the first time, the first basic block in the CFG, a type test, is compiled, two compilation stubs are created, one is triggered if the test succeeds, the other is triggered if the test fails. The property *type of n is fixnum* is attached to the first stub and the property *type of n is unknown* is attached to the second stub. Note that it could be possible to attach the property *type of n is not fixnum* to the second stub. However, because this property does not enable immediate optimization and to simplify the explanation, we consider the type is still unknown at this point. In our implementation presented Section 5, we have not explored the propagation of this property either.

Figure 1.3 shows the compilation state after the first block is compiled. We can see in the dotted boxes the discovered properties the compiler uses to specialize the generated code (the specialization context). Because the type of `n` is initially unknown, the first version of basic block 1 is generic. The two compilation stubs are then created with an attached context to handle both branches, compilation suspends and the code generated for block 1 is executed.

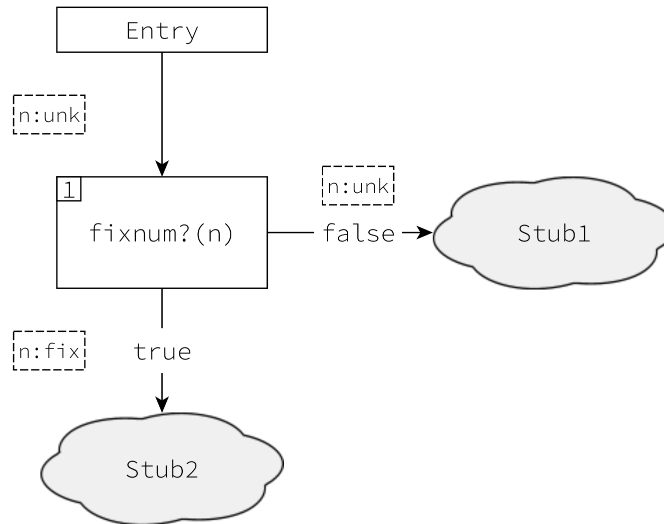


Fig. 1.3. Compilation state after the first block is generated.

Assuming `n` is a fixnum, the stub `Stub2` is executed, it calls the compiler, and the following block (block 5) is specialized using the information gained from the stub. Using the same process, the rest of the blocks are specialized using the gained information. When block 7 is reached, no test is generated because the compiler knows, from the propagated information, that at this point `n` is a fixnum thus control flows unconditionaly to block 10 and block 10 is generated.

Figure 1.4 shows the compilation state after the `abs` function has been called with a fixnum parameter. We see that using BBV, the type of `n` is propagated, blocks 5 and 10 are specialized for `n` being a fixnum thus a single type test is executed for each call to `abs` with a fixnum (no code is generated for block 7). If the function is later called with a flonum parameter, the stub `Stub1` is triggered and, using the same mechanism, the type of `n` is discovered, propagated and the rest of the blocks are specialized using this type.

Figure 1.5 shows the compilation state after the function has been called with `n` being a negative fixnum and a positive flonum. As we can see, the type of `n` is propagated from the first test, and redundant tests are removed. A single type dispatch (a fixnum and a flonum type tests) is executed at the entry point of the function resulting in two specialized versions of the function body. In addition, with the lazy design of BBV, only executed paths are generated and specialized.

To build a JIT compiler based on BBV, two design constraints are then required:

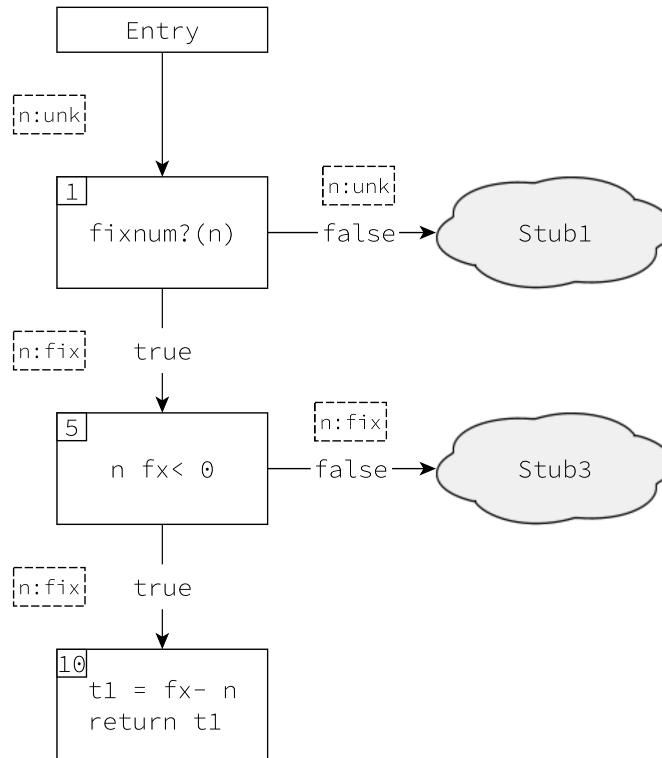


Fig. 1.4. Compilation state after the whole function is generated for a call with a negative fixnum.

(1) It must use a lazy incremental compilation design. Each time the compiler detects that it could discover properties depending on the control flow path, code generation must be delayed until one of the paths is executed.

(2) It must be able to specialize the generated code. Multiple versions of the basic blocks may need to be generated depending on the information available in the compilation contexts.

1.1.6. Scheme

In this thesis, we will use Scheme to illustrate our ideas. Scheme is a programming language created in the 1970s by Guy Lewis Steele and Gerald Jay Sussman [**134**]. It is a language highly inspired by Lisp and today considered as one of the two main dialects of Lisp. Scheme follows a minimalist design allowing quick prototyping thus making it popular in academic research and among programming language implementers to explore new techniques and optimizations. Scheme is a multi-paradigm programming language offering functional and imperative programming. It is a dynamic language, it offers dynamic typing, hygienic

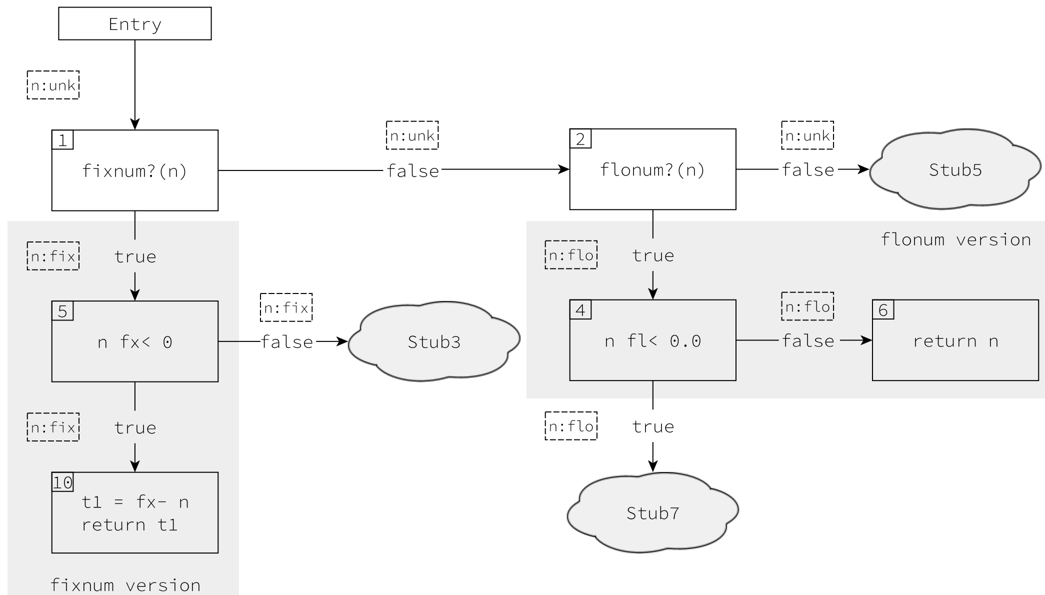


Fig. 1.5. Compilation state after the whole function is generated for a call with a negative fixnum and a positive flonum.

macros, *eval* and homoiconicity. It is also a higher-order language offering first class functions and continuations (i.e. functions and continuations can be passed as function arguments and returned by functions).

Several reasons motivate us to choose Scheme for our experiments. Thanks to its minimalist design, it allows quickly building an implementation supporting a subset of the language features to test and evaluate techniques and optimizations. Several basic forms of Scheme can be implemented with macros using other basic forms. For example, a Scheme expression using the basic form *let** can easily be transformed by a macro to another Scheme expression using a *let* form with the same semantics that can itself be transformed to another expression using *lambda* expressions with the same semantics. Using this approach, the implementation requires to implement a limited set of basic forms. This approach allows quick prototyping but may suffer from poor performance. For instance, the Scheme *let* and *let** forms are used to create local variable bindings. If they are naively expanded to *lambda* expressions, a function call is generated for each *let* and *let** form. It is possible to improve the performance of the Scheme implementation using an incremental approach. To generate more efficient code, native support of basic forms (or equivalently detecting special patterns of basic forms) can be added to the implementation one after the other. The *let* form is typically handled specially

by implementations to avoid the closure creation and function call overhead. Moreover, this incremental approach allowed us to focus on critical parts of the implementation.

Another reason we use Scheme is that when we started our research there was no optimizing JIT compiler for Scheme, not to mention tracing JIT compilation (presented in Section 2.2.1). No study existed on how modern JIT compilation techniques can be applied to the specific set of features Scheme offers and how they impact performance in this context. For instance, generating efficient code with interprocedural optimizations in the presence of tail call optimization, higher-order functions and dynamic typing is a problem that has been widely explored by most of the Scheme implementations through static techniques that are suitable for use in AOT compilers (more details on these techniques are given in Section 3.2). No study shows how this specific problem can be addressed in a JIT context yet JIT compilation opens up opportunities for new techniques and optimizations.

Regarding development effort, Clinger and Hansen [**40**] presented an implementation of the Scheme programming language, Twobit, with relatively good performance with a focus on simplicity and development effort. However, that study focuses on AOT compilation and no similar study exists for JIT compilation of Scheme.

In this thesis, in addition to explaining lazy BBV extensions which are applicable to any language, we address these specific issues showing how simple JIT compilation techniques impact the performance of a language offering a set of features similar to Scheme, in particular higher-order functions.

Chapter 2

Compiler Design for Extremely Lazy Specializing Compilation

Part of the work presented in this chapter is published in the two papers:

- Baptiste Saleil and Marc Feeley. Code Versioning and Extremely Lazy Compilation of Scheme. In *Scheme and Functional Programming Workshop, 2014.* [**112**]
- Baptiste Saleil and Marc Feeley. Building JIT Compilers for Dynamic Languages with Low Development Effort. In *Proceedings of the 10th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages, VMIL, 2018.* [**116**]

In this chapter, we expand on the ideas presented in those papers.

2.1. Introduction

Building high performance multi-tier JIT VMs requires significant development effort. We argue that building multi-tier VMs is not suited to context in which development resources are limited and relatively good performance is needed. As previous research showed [**37**], BBV is attractive for baseline compilers (first compilation level) in multi-tier VMs because it allows achieving relatively good performance without requiring high compilation time. Thanks to its simplicity, BBV can be used in a single-tier VM to offer a good trade-off between performance and development effort. It could also replace the lower tiers of a multi-tier VM, including the interpreter. However, as explained in Section 1.1.5, BBV uses a basic block based IR to specialize the code.

Consequently, to use BBV, the compiler first needs to parse the source program and transform it to an Abstract Syntax Tree (AST). Once the AST is built, it is transformed

into a CFG based IR with basic blocks. Then, BBV is applied lazily on the IR to generate specialized code in the target language.

In this chapter, we present how we have adapted BBV to work directly from the AST allowing building compilers using a single compilation level with a single internal representation of the source program, the AST.

2.2. Related Work

2.2.1. Tracing

Research has shown that interpreters are considerably easier to build than typical JIT compilers. Once the AST of a program is generated, it can easily be interpreted using a tree walking interpreter. However, performance is significantly impacted compared to the execution of machine code. The DynamoRIO [**17, 133**] project has shown that the negative impact on performance of the interpretation of dynamic languages can be decreased using compilation traces (commonly referred to as *tracing*). Tracing was then adapted to JIT compilation of high-level languages in HotpathVM [**57**]. This compilation approach has been used to remove type tests and to reduce run time work [**56, 128, 23**]. When a compiler uses tracing, it first executes the program using an interpreter. When it detects a frequently executed loop (*hot* loop), it records the instructions executed on the next iteration of the loop, following function calls, and stores these instructions using a linear sequence of IR instructions (the trace). Typical optimizations are then applied to the trace and optimized code is generated and executed. Because traces have no internal control-flow joins, the IR is simplified and the development effort is decreased. Several compilers implement tracing JIT compilation for various languages [**17, 133, 56, 95**] including Scheme [**18**]. These systems have demonstrated good performance. Even if it is simplified, the compiler still needs to use an IR between the AST and the generated code. In addition, to detect hot loops and profile code before generating target code, tracing essentially requires both the use of an interpreter and a compiler. Building, maintaining and guaranteeing equivalent semantics of two execution systems increases development effort.

2.2.2. Language Implementation Frameworks

Other research has shown that to achieve relatively good performance with low development effort, we can instead build interpreters and focus on optimizing the interpreter itself instead of the program it executes. The RPython project [**109, 24**] is a framework allowing automatically generating JIT compilers. To generate a JIT compiler, an interpreter is first built using the RPython language and the framework generates the JIT compiler from this interpreter. The particularity of RPython is that it uses tracing compilation on the interpreter instead of the executed program. This technique is commonly referred to as *meta-tracing* [**22**]. To implement a language with relatively good performance, a single interpreter can be built, significantly decreasing the development effort. This framework has for example been used to build JIT compilers for Python [**109, 24**] and Racket [**18, 19**], a dialect of Scheme. Both projects show that RPython can be used to generate efficient language implementations.

The Truffle project [**145, 144**] also focuses on decreasing language implementation effort. This framework also allows building an efficient implementation by writing an interpreter only. The system then optimizes this interpreter using type feedback and other profiling information. Unlike RPython, Truffle uses a traditional method-based JIT compiler [**141**]. Truffle is used to build several dynamic language implementations such as Python [**138**], JavaScript [**70**] and Ruby [**118**]. These systems have demonstrated good performance.

These projects offer a significant step forward in programming language implementation with low development effort. However, these generic frameworks give a limited control over the compilation phases, particularly for low level techniques and implementation. For example, it may not be possible to choose the run time representation of values, memory management algorithms, the applied optimizations and the set of supported target machines. These frameworks are thus relevant mainly when full control is not needed. Note that this may be hard to know when starting a language implementation effort, and at a later stage there is too much invested in the framework to change if full control becomes needed. In addition, if hardware resources are limited, for example with embedded systems, simple and lightweight techniques are generally preferred over complex generic frameworks.

2.2.3. Generic Intermediate Representations

An alternative to decrease development effort is to use an existing VM based on an explicitly represented IR (commonly referred to as *bytecode*) such as LLVM [**85**], the Java Virtual Machine (JVM) [**86**] and the Common Language Runtime [**92**]. These systems are able to execute programs written in the IR specially designed for them. To implement a language L, source programs written in language L are translated to the IR of the target system. The main advantage of using a bytecode based system is that the language implementation does a single AOT compilation pass to generate the IR and completely relies on the optimizations and the execution environment of the VM. This significantly decreases the required development effort. However, these systems raise the same issues as the frameworks we presented above. They give a limited control over the low level optimizations and techniques, and add an important software dependence.

In addition, it is possible that language L offers features that are not natively supported by these systems. Workarounds are often used to implement missing features on top of these systems causing a negative impact on the maintainability and overall performance of the implementation. These features include Tail Call Optimization (TCO), the run time representation of values (e.g. Tagging and NaN-boxing), weak references and low-level number implementations such as fixnums and bignums. For example, TCO is not available on the JVM. Because TCO is required by Scheme, Scheme implementations built on top of the JVM use workarounds or limit the use of TCO [**119, 26**].

2.3. Building and Executing Abstract Syntax Trees

Building ASTs from source programs is a process that has been studied for many years. Today, there are several existing solutions to build ASTs from source programs efficiently, making this transformation easy to implement without requiring significant development effort. Depending on the nature of the grammar of the source language, specific tools can be used to build ASTs. Strings of LL(k) grammars can easily be parsed and transformed to ASTs using *recursive descent* algorithms [**8**]. Parser generators such as ANTLR [**103**] and Flex/Bison [**104, 45**] can also be used to parse LL(k), LALR and LR grammars. These tools automatically generate parsers from formal grammar specifications. For the rest of this

chapter, we assume the compiler is already able to build ASTs from source programs and will not address this issue further.

Executing a program represented by an AST can easily be done with a tree walking interpreter. The nodes of the AST, starting from the root node, are visited in the order they need to be executed. Unfortunately performance is typically poor. A quick comparison of a program computing the 30th Fibonacci number with a naive tree walking interpreter and the Gambit AOT Scheme compiler [**51**] shows a 300× performance gap.

AOT code generation from an AST often offers better performance, even if the compilation is performed directly from the AST, without using code transformation and optimizations on IRs. This approach is referred to as *naive AOT compilation* in the rest of this thesis. Naive AOT compilation can also be easily implemented using a tree walker. The nodes of the AST are recursively visited, starting from the root node. For each visited node, the compiler generates a piece of code in the target machine language following the control flow. This naive AOT approach improves performance compared to interpretation but, unlike JIT compilation, does not enable optimizations necessary to efficiently implement dynamic languages.

Using JIT compilation typically involves adding transformations to use program representations more suitable to JIT compilation techniques. For example, BBV requires the source program to be represented by a CFG of basic blocks and Tracing requires it to be represented by traces.

The technique we present in this chapter consists of implementing JIT compilation using an approach similar to naive AOT code generation from ASTs. This means that the compiler generates code directly from ASTs using a recursive tree walking algorithm. The technique does not require the use of a low level IR relying on code transformation from the AST. We explain in Chapter 5 that our implementation uses some code transformations on the AST, but no transformation to intermediate low level representation is performed. We first present how the code is typically generated by naive AOT compilers using AST walkers. Then, we explain how these tree walkers can be extended to enable JIT compilation and BBV without significantly increasing development effort.

$$
\begin{array}{lcl}
\langle E \rangle & ::= & \langle \mathit{Constant} \rangle \\
& | & \langle \mathit{Id} \rangle \\
& | & \textbf{(println } \langle E \rangle \textbf{)} \\
& | & \textbf{(let } \langle \mathit{Id} \rangle\ \langle E \rangle\ \langle E \rangle \textbf{)} \\
& | & \textbf{(if } \langle E \rangle\ \langle E \rangle\ \langle E \rangle \textbf{)} \\
\langle \mathit{Id} \rangle & ::= & [\textbf{a-z}]+ \\
\langle \mathit{Constant} \rangle & ::= & \langle \mathit{Fixnum} \rangle \\
& | & \langle \mathit{Boolean} \rangle \\
\langle \mathit{Fixnum} \rangle & ::= & [\textbf{0-9}]+ \\
\langle \mathit{Boolean} \rangle & ::= & \#\textbf{t} \\
& | & \#\textbf{f}
\end{array}
$$

Fig. 2.1. Grammar of a simple language, based on S-expressions, used as an example in this chapter.

2.4. Source Language Example

Figure 2.1 shows the grammar of a simple language based on S-expressions that is used in the rest of this chapter to illustrate our approach. This language allows local variable bindings (`let` form) and allows printing values (`println` form). It also supports constants and a conditional construct (`if` form). We explain how the code is generated for this simple language, but it is straightforward to extend it to other constructs commonly used by dynamic languages.

2.5. Abstract Target Machine

To simplify the explanation of the code generation process in this chapter, we use the abstract execution machine defined in Figure 2.2 as the target machine. This abstract machine is stack-based, it uses 10 instructions that can easily be implemented in real machines.

The instruction stream is represented by a list of instructions. The `code-stream` variable contains a pointer to the beginning of the stream. The `code-stream-end` variable contains a pointer to the end of the stream (last instruction generated). The `gen` function adds the given

```
1:  (define code-stream (list 'main))
2:  (define code-stream-end code-stream)
3:
4:  (define (gen . instrs)
5:    (set-cdr! code-stream-end instrs)
6:    (set! code-stream-end (last-pair instrs)))
7:
8:  (define (dest label) (member label code-stream))
9:
10: (define (start-execution label stack) (exec (dest label) stack))
11:
12: (define (exec pc s)
13:   (match (car pc)
14:     ((push-lit ,v) (exec (cdr pc) (cons v s)))
15:     ((push-loc ,i) (exec (cdr pc) (cons (list-ref s i) s)))
16:     ((swap)        (exec (cdr pc) (cons (cadr s) (cons (car s) (cddr s)))))
17:     ((pop)         (exec (cdr pc) (cdr s)))
18:     ((add)         (exec (cdr pc) (cons (+ (car s) (cadr s)) (cddr s))))
19:     ((println)     (println (car s)) (exec (cdr pc) (cons #f (cdr s))))
20:     ((jump ,d)     (exec (dest d) s))
21:     ((iffalse ,d)  (exec (if (eqv? (car s) #f) (dest d) (cdr pc)) (cdr s)))
22:     ((halt)        s)
23:     ((callback ,t) (set-car! pc `(jump ,(t))) (exec pc s))
24:     (,label        (exec (cdr pc) s))))
```

Fig. 2.2. Abstract stack-based target machine definition.

instructions to the stream. These instructions are added at the end of the stream meaning that only the `code-stream-end` variable has to be updated.

The `exec` function executes the instruction stream. The `match` special form pattern matches the value against a set of patterns to choose a branch. It is similar to a `case` special form but the patterns can contain variables prefixed with commas.

There are 4 instruction groups in this abstract machine:

- **Stack management instructions:** The instructions `push-lit`, `push-loc`, `swap` and `pop` respectively allow to push a constant on top of the stack, copy and push a slot of the stack, swap the two values on top of the stack and remove the value on top of the stack.
- **Data instructions:** The instruction `add` adds the two values on top of the stack and the instruction `println` prints the value on top of the stack.

- **Control flow instructions:** The instructions `jump`, `iffalse` and `halt` respectively allow to unconditionally jump to a label, jump to a label if the value on top of the stack is false and stop the execution.
- **Callback instruction:** The `callback` instruction allows implementing a callback mechanism to return execution to the compiler (more details on this instruction are given in the next sections). Note that it is a self modifying instruction, replacing itself by a jump instruction.

Because this abstract machine is simple and the instructions are conceptually close to the instructions of existing machines, the code generation examples given in this chapter can easily be transformed to target real machines.

2.6. Code Generation from Abstract Syntax Trees

A typical approach to generate code from ASTs is to use a tree walker to visit each node and generate a code sequence for it. Figure 2.3 shows a typical tree walker used to generate code from an input S-expression by matching each type of sub-expression. In this way, the code can be generated from an AST using a single compilation pass.

If a constant or a variable is matched, the code to push the value is generated. For the `println` form, the compiler generates the `println` instruction after generating code for the expression representing the value to print (`E`).

For the `let` form, it first generates the code for the binding expression (`E1`). It then generates code for the `let` body (`E2`) with the environment extended with the new binding. Finally, it generates code to clean the stack after the `let`.

For the `if` form, the compiler generates code for the condition (`E1`), then it generates the control flow instructions to dispatch the execution to the branches using labels it created. The two branches (`E2` and `E3`) are then generated.

Note that in the `gen-program` function, the compiler generates the final `halt` instruction after generating the whole program.

This approach allows building compilers generating machine code, significantly improving performance compared to interpreters, without significantly increasing development effort. However, because naive AOT compilers cannot customize the generated code using the

```
 1: (define (gen-expr expr env)
 2:
 3:   (match expr
 4:
 5:     (,c when (constant? c)
 6:      (gen `(push-lit ,c)))
 7:
 8:     (,v when (variable? v)
 9:      (gen `(push-loc ,(index-of v env))))
10:
11:     ((println ,E)
12:      (gen-expr E env)
13:      (gen `(println)))
14:
15:     ((let ,v ,E1 ,E2) when (variable? v)
16:      (gen-expr E1 env)
17:      (gen-expr E2 (cons v env))
18:      (gen `(swap) `(pop)))
19:
20:     ((if ,E1 ,E2 ,E3)
21:      (gen-expr E1 env)
22:      (let ((join (gensym 'join))
23:            (else (gensym 'else)))
24:        (gen `(iffalse ,else))
25:        (gen-expr E2 env)
26:        (gen `(jump ,join))
27:        (gen else)
28:        (gen-expr E3 env)
29:        (gen join)))
30:
31:     (else (error "Unknown expression"))))
32:
33: (define (gen-program body)
34:   (gen-expr body '(arg))
35:   (gen `(halt)))
```

Fig. 2.3. Typical pattern matching used to generate code for an input S-expression.

dynamic properties of the program, performance is still far from ideal when this approach is used to implement languages compared to JIT compilation.

2.7. Lazy Compilation from Abstract Syntax Trees

As explained previously, a lazy JIT compilation design is required to implement lazy BBV. Depending on the path control takes at execution, lazy compilation allows the compiler to discover the properties used to specialize the code.

To add laziness with minimal added complexity, the compilation process can be expressed using Continuation-Passing-Style (CPS) [**134**].

A continuation argument is added to the function generating the code for a node. The continuation is a function that, when called, generates the code following the code that must be generated for the currently visited node. Consequently, it is the continuation of the code generation process for the current node.

When the compiler starts visiting the tree from the root node, it passes the initial continuation that generates the code following the whole program (the final `halt` instruction).

A benefit of expressing the compilation process using CPS is that, when a node dispatching the control flow is visited, the compiler can trivially suspend code generation as the code generation state is already packaged in the continuation. Instead of generating the code for the branches, it generates callback instructions returning control to the compiler. These callbacks contain the continuation of each branch. When the dispatch is executed, the appropriate callback is executed and the compiler is invoked with the continuation and the outcome of the dispatch to resume compilation.

2.7.1. Implementation of the Callback Mechanism

Figure 2.4 shows how the callback mechanism can be implemented on top of the abstract machine. The `gen-stub` function is responsible for generating compilation stubs. A stub is implemented with a `callback` instruction which contains the `thunk` function that calls back the compiler. When the abstract machine executes this instruction, it retrieves the thunk function and calls it.

The `gen-expr-lazily` function allows generating a jump (conditional or unconditional) to a branch that is not yet generated and must be generated lazily (i.e. when the execution flows to it). To implement this mechanism, a stub is created using the `gen-stub` function with a newly created callback. This callback calls the compiler to generate the branch using the continuation `cont` and the environment `env` given to the `gen-expr-lazily` function.

```
 1: (define (gen-stub thunk)
 2:   (let ((label (gensym 'stub)))
 3:     (set!
 4:       code-stream
 5:       (cons label (cons `(callback ,thunk) code-stream)))
 6:     label))
 7:
 8: (define (gen-expr-lazily jump expr env cont)
 9:   (let ((instr `(,jump ???)))
10:     (set-car!
11:       (cdr instr)
12:       (gen-stub (lambda ()
13:                   (let ((label (gensym 'lazily_generated)))
14:                     (set-car! (cdr instr) label)
15:                     (gen label)
16:                     (gen-expr expr env cont)
17:                     label))))
18:     (gen instr)))
```

Fig. 2.4. Implementation of the callback mechanism on top of the abstract machine.

Once the branch is generated, the jump to the stub can be patched to directly jump to the generated branch instead on subsequent executions.

Note that the `gen-expr-lazily` function adds the generated code at the end of the stream. However, the `gen-stub` function adds the stubs at the beginning of the stream to simulate an implementation in which the instructions and the stubs are generated in two separate spaces (as is the case in our implementation).

On real machines, the mechanism can be implemented in a similar way. A stub is typically implemented by a native call to a function of the compiler. The identity of the callback and the position of the jump that must be patched can be given as an argument to this call. Once a branch is generated, the stub can be collected. Memory management for the stubs is relatively simple because all stubs are the same size and they are typically unreachable and reclaimable after being executed.

2.7.2. Implementation of Continuation-Passing Style Code Generation

Figure 2.5 shows how the `gen-expr` function is extended to do lazy compilation using CPS. The extra argument `cont` is the code generation continuation of the S-expression `expr` currently matched.

```
 1: (define (gen-expr expr env cont)
 2:
 3:   (match expr
 4:
 5:     (,c when (constant? c)
 6:      (gen `(push-lit ,c))
 7:      (cont))
 8:
 9:     (,v when (variable? v)
10:      (gen `(push-loc ,(index-of v env)))
11:      (cont))
12:
13:     ((println ,E)
14:      (gen-expr E env (lambda () (gen `(println)) (cont))))
15:
16:     ((let ,v ,E1 ,E2) when (variable? v)
17:      (let* ((exit (lambda ()
18:                     (gen `(swap) `(pop))
19:                     (cont)))
20:             (body (lambda ()
21:                     (gen-expr E2 (cons v env) exit))))
22:        (gen-expr E1 env body)))
23:
24:     ((if ,E1 ,E2 ,E3)
25:      (let ((test (lambda ()
26:                    (gen-expr-lazily 'iffalse E3 env cont)
27:                    (gen-expr-lazily 'jump E2 env cont))))
28:        (gen-expr E1 env test)))
29:
30:     (else (error "Unknown expression"))))
31:
32: (define (gen-program body)
33:   (gen-expr body '(arg) (lambda () (gen `(halt)))))
```

Fig. 2.5. Typical pattern matching used to generate code for an input S-expression with lazy compilation.

If a constant or a variable is matched, the code to push the value is generated. The continuation is then called immediately to continue code generation.

For the `println` form, the compiler recursively generates the code for the expression representing the value to print (`E`) by calling the `gen-expr` function. It passes to `gen-expr` a continuation generating the `println` instruction and calling the continuation.

For the `let` form, the compiler first creates two continuations. The continuation `exit` generates code to clean the stack after the `let`. The continuation `body` generates the body of the `let` (`E2`) and updates the environment with the new binding. To generate the body, the compiler calls the `gen-expr` function and gives it the `exit` continuation. Finally, the compiler generates code for the variable expression (`E1`) with the `body` continuation.

The `if` form uses lazy code generation. For this form, the compiler first creates the continuation `test`. This continuation generates the code to test the result of the first expression and dispatch the execution to branches that are not yet generated. Two jumps (`iffalse` and `jump`) are generated by the test continuation using the callback mechanism. They are then generated through the `gen-expr-lazily` function using:

- The expression of the branch the jump is associated with (`E2` or `E3`)
- The current environment (`env`)
- The current continuation (`cont`)

Once the continuation generating the dispatch is created, the compiler generates the code for the first expression using the `gen-expr` function by giving it the `test` continuation. When the dispatch code is generated, the compilation process is suspended. A branch is only generated from the callbacks when the dispatch is executed.

It is worth mentioning that it is possible that the code generation is suspended before the dispatch is generated. This is for instance the case if the condition of the `if` contains another `if`. In this situation, the code generation process is first suspended when generating the dispatch of this `if` because it is the first in the execution flow. The process will then be suspended again when the second dispatch is generated.

Note that with this implementation, in the case the two branches of the `if` are executed, the continuation `cont` is called two times meaning that the code is generated twice, resulting in tail duplication. This tail duplication can be avoided by generating the tail first (either eagerly or first as a stub) and passing to the branches a continuation generating a jump to

the tail. However, we explain in the next section how code specialization naturally addresses this issue.

In the `gen-program` function, the compiler calls the `gen-expr` function with the AST root node to generate code for the program. It passes to this function the initial continuation that generates the final `halt` instruction.

This indicates that representing the compilation process using continuations allows building JIT compilers using a simple compilation design based on an AST walker which is similar to naive AOT compilation of an AST. With this design, the JIT compiler is able to delay code generation and to compile each node separately, without requiring knowledge of control or data flow generally collected using relatively expensive control and data flow analysis on CFGs.

Although continuation chains can be considered as an intermediate representation, the implementation is close to a classical tree walker and does not imply code transformation from the AST to another representation. This design allows JIT compilation of the source program directly from the AST, without requiring the use of such an IR.

As presented in this section, the design does not apply any optimization typically used by compilers. We present in the next sections how they can easily be applied on top of this design.

2.8. Code Specialization from Abstract Syntax Trees

To implement BBV from the AST, it is necessary to add to the compiler the ability to specialize the code generated by the continuations according to a compilation context.

A parameter is added to the continuations in the CPS representation to represent the specialization context. This context contains what is known of the dynamic properties of the program at that point of the execution and is used by the continuation to specialize the code it generates. The context is used in two ways:

(1) **property addition.** Before calling the successor continuation, the compiler adds to the context the properties it discovered when generating code for the currently executed continuation.
(2) **property use.** When generating code for a continuation, the compiler uses the available properties from the context to optimize the generated code.

As an example, consider the case where the compiler uses type information to specialize the code generated by the continuation. When the compiler discovers the type of a value, it *adds* it to the context and passes it to the following continuations. These continuations can then *use* this property to generate more efficient code, such as removing redundant type tests.

The compiler must also be able to cache the code generated by the continuations. When a continuation needs to be specialized for a given context, it first checks if this version exists in the cache. If the version does not exist, it is generated, added to the cache, and compilation continues. If the version exists, a jump to the existing version is generated instead, and compilation stops.

2.8.1. Implementation of the Contexts and Version Caching

Figure 2.6 shows how the contexts can be represented for type properties and how the caching mechanism can be implemented on top of the abstract machine. In this example, the context is represented by a stack of types. Each slot of this stack is a type of a slot of the execution frame. Four operations are added to handle contexts:

- **cpush:** Add the given type on top of the context's stack
- **ctop:** Return the type on top of the context's stack
- **cpop:** Remove the type on top of the context's stack
- **cref:** Return the type at the given index of the context's stack

Version caching is implemented here using association lists for simplicity. The global `versions` variable is a list of key-value pairs. The key is the identity of the continuation combined with the context used to specialize the code it generates. The value is the label of the specialized version in the generated code.

The `get-cont-label` function checks if a version specialized for the `ctx` context exists for the `cont` continuation. Note that in this example, to check if a version exists in the cache, the `assoc` function is used with a pair containing the continuation and the context. We assume that the Scheme implementation compares the identity of the continuations and compares all types of the contexts. When the version exists, it calls the `existing` function. When the version does not exist, it generates it inline and adds it to the cache. We can see that when the version does not exist, the `cont` continuation is called to generate the code and the specialization context is given as an argument.

```
1:  (define (cpush ctx type) (cons type ctx))
2:  (define ctop car)
3:  (define cpop cdr)
4:  (define cref list-ref)
5:
6:  (define versions '())
7:
8:  (define (get-cont-label cont ctx existing)
9:
10:   (define (gen-version cont ctx label)
11:     (gen label)
12:     (cont ctx))
13:
14:   (let ((version (assoc (cons cont ctx) versions)))
15:     (if version
16:         (existing (cdr version))
17:         (let ((label (gensym 'v)))
18:           (gen-version cont ctx label)
19:           (set! versions
20:                 (cons (cons (cons cont ctx) label) versions))
21:           label))))
22:
23: (define (gen-cont cont ctx)
24:   (get-cont-label cont ctx (lambda (label)
25:                              (gen `(jump ,label)))))
```

Fig. 2.6. Implementation of the contexts and version caching.

The `gen-cont` function generates the code for a given continuation specialized for the given context. This function uses the `get-cont-label` function to generate the specialized version or a jump to the version if it exists.

Note that to prevent code explosion in case of over specialization, this code can easily be extended to set a limit *lim* in the number of versions. When generating a new specialized version for a continuation, if *lim* specialized versions have already been generated, the compiler can instead fallback to a generic version or to an existing version that is more generic than the one the compiler must generate.

Using version caching, the issue of tail duplication at `if` join points is eliminated. If the two branches of a `if` are executed and the control flows to the join point with the same context, the version of the join point for this context is generated when it is reached the first

time and this version is reused when it is reached the second time. Tail duplication only occurs if the join point is reached with different contexts thus if it is specialized.

2.8.2. Implementation of Code Specialization

To implement code specialization in the tree walking compiler, the specialization context must be added as an argument to the continuations, as shown in Figure 2.7. When the compiler must generate the code of a continuation, it uses the `gen-cont` function to use the caching mechanism instead of directly calling the continuation. Finally, code generation in the continuation is updated to use the properties available in the context to generate more optimized code and to add the newly discovered properties to the context.

If a constant is matched, the code to push the constant is generated and its type is added to the context (this is an example of *property addition*). The successor continuation is then called to generate or reuse the version specialized for this new context.

If a variable is matched, the code to push it is generated, the compiler retrieves its type from the context, adds it on top of the context's stack and gives the context to the successor continuation.

For the `println` form, the compiler first generates the argument expression using the current context. In the continuation, the compiler first generates the `println` instruction, pops the type from the context, adds the type of the result of the `println` instruction to the context's stack and gives the context to the successor continuation.

For the `let` form, the compiler first generates the code for the value expression using the current context. The `body` continuation does not change the context it receives. At this point, the type of the value expression is already on top of the context's stack. In the `exit` continuation, the compiler retrieves the type of the value returned by the let, updates the context to remove the type of the variable and gives the context to the successor continuation.

For the `if` form, the compiler first generates the code for the condition using the current context. The `test` continuation shows an example of *property use*. When this continuation is called to generate code specialized for a given context, the compiler first retrieves the propagated type of the condition. When the type is *boolean* or *unknown*, the result of the test is not known at compile time and the compiler must generate the dispatch code. When the type is known and is not *boolean*, it can't be false so it is not necessary to generate the

```
 1: (define (gen-expr expr env ctx cont)
 2:
 3:   (match expr
 4:
 5:     (,c when (constant? c)
 6:      (gen `(push-lit ,c))
 7:      (gen-cont cont (cpush ctx (if (number? c) 'num 'bool))))
 8:
 9:     (,v when (variable? v)
10:      (let ((i (index-of v env)))
11:        (gen `(push-loc ,i))
12:        (gen-cont cont (cpush ctx (cref ctx i)))))
13:
14:     ((println ,E)
15:      (gen-expr E env ctx (lambda (ctx)
16:                            (gen `(println))
17:                            (let ((ctx (cpush (cpop ctx) 'bool)))
18:                              (gen-cont cont ctx)))))
19:
20:     ((let ,v ,E1 ,E2) when (variable? v)
21:      (let* ((exit (lambda (ctx)
22:                     (gen `(swap) `(pop))
23:                     (let* ((type (ctop ctx))
24:                            (ctx (cpush (cpop (cpop ctx)) type)))
25:                       (gen-cont cont ctx))))
26:             (body (lambda (ctx)
27:                     (gen-expr E2 (cons v env) ctx exit))))
28:        (gen-expr E1 env ctx body)))
29:
30:     ((if ,E1 ,E2 ,E3)
31:      (let ((test (lambda (ctx)
32:                    (let ((type (ctop ctx)))
33:                      (if (member type '(bool unknown))
34:                          ;; outcome is unclear so generate a run time test
35:                          (begin
36:                            (gen-expr-lazily 'iffalse E3 env (cpop ctx) cont)
37:                            (gen-expr-lazily 'jump E2 env (cpop ctx) cont))
38:                          ;; outcome is known to be trueish
39:                          (begin
40:                            (gen `(pop))
41:                            (gen-expr E2 env (cpop ctx) cont)))))))
42:          (gen-expr E1 env ctx test)))
43:
44:     (else (error "Unknown expression"))))
45:
46: (define (gen-program expr)
47:   (lambda (ctx)
48:     (gen-expr expr '(arg) ctx (lambda (ctx)
49:                                 (gen `(halt))))))
```

Fig. 2.7. Lazy compilation and code specialization for an input S-expression.

dispatch. The compiler knows that when this version will be executed the condition will succeed. It can then directly generate the true branch inline. Before generating the branches, the compiler removes the type of the condition from the context.

Note that because the `ctx` argument is added to the `gen-expr` function it must also be passed to the `gen-expr-lazily` function when using the callback mechanism. The `gen-expr-lazily` function is extended to use it when calling the `gen-expr` function.

2.9. Example of Type Test Removal

We explained in Section 1.1.5 that to ensure the safety of the primitives, typical Scheme compilers generate type tests at primitives to check that the type of the operands is appropriate for the primitive and raise an error otherwise. BBV is efficient at removing these type tests and because our approach allows the implementation of BBV from the AST, it also allows removing these tests.

Figure 2.8 shows the grammar extended with fixnum type tests (`fixnum?`), the addition operator on fixnums (`fx+`) and the `error` expression. We assume that the compiler uses an expansion phase before executing the code of a program. In this expansion phase, the compiler transforms each call to the + function to fixnum type tests (`fixnum?`) on the operands followed by a fixnum addition (`fx+`) in case the tests succeed or an error in case a test fails. Figure 2.9 shows an example of expansion.

The `fixnum?` type test form can be implemented in a similar way to the `if` form. When the type test continuation is called (the `test` continuation), the compiler first determines if the type of the variable is known in the context. Three cases may occur:

(1) The type is known and is *fixnum*. No `fixnum?` type test is generated and the *then* branch is generated inline.

(2) The type is known and is not *fixnum*. No `fixnum?` type test is generated and the *else* branch is generated inline.

(3) The type is unknown. The `fixnum?` type test is generated.

In the last case (unknown type), the context given to the `gen-expr-lazily` function for the true branch is updated to reflect the type discovered of the tested variable. When the test is executed, if it succeeds, the true branch is lazily generated and specialized using this newly

$$
\begin{array}{lcl}
\langle E \rangle & ::= & \langle \mathit{Constant} \rangle \\
 & | & \langle \mathit{Id} \rangle \\
 & | & \textbf{(println } \langle E \rangle \textbf{)} \\
 & | & \textbf{(let } \langle \mathit{Id} \rangle\ \langle E \rangle\ \langle E \rangle \textbf{)} \\
 & | & \textbf{(if } \langle E \rangle\ \langle E \rangle\ \langle E \rangle \textbf{)} \\
 & | & \textbf{(if (fixnum? } \langle \mathit{Id} \rangle \textbf{))}\ \langle E \rangle\ \langle E \rangle \textbf{)} \\
 & | & \textbf{(error)} \\
 & | & \textbf{(fx+ } \langle \mathit{Id} \rangle\ \langle \mathit{Id} \rangle \textbf{)} \\
\langle \mathit{Id} \rangle & ::= & \textbf{[a-z]+} \\
\langle \mathit{Constant} \rangle & ::= & \langle \mathit{Fixnum} \rangle \\
 & | & \langle \mathit{Boolean} \rangle \\
\langle \mathit{Fixnum} \rangle & ::= & \textbf{[0-9]+} \\
\langle \mathit{Boolean} \rangle & ::= & \textbf{\#t} \\
 & | & \textbf{\#f}
\end{array}
$$

Fig. 2.8. Grammar of a simple language with variables, constants and if.

```
                    (if (fixnum? a)
                        (if (fixnum? b)
(+ a b)     ⇒               (fx+ a b)
                            (error))
                        (error))
```

Fig. 2.9. Example of code expansion to insert type tests for primitives.

discovered type. Consequently, this type will be propagated to the following continuations and the rest of the code will be specialized using this property.

To fully take advantage of the discovered type information, the specialization context can be extended to include aliasing information. For each variable, the compiler must keep track of the copies of the variable (its positions in the context's stack). When the type of a variable is discovered, the compiler changes the type in all the slots.

In the same way that BBV applied to typing allows removing type tests using a lazy compilation design from an IR, our approach allows propagating the discovered properties through code specialization and allows removing type tests.

Using our approach to compile the function `abs` (with suitable extensions to the compiler to support the `negative?` and `-` primitives), the number of generated and executed type tests is the same as when classical BBV is used. Once the type of the argument `n` is discovered for the `negative?` primitive, it is propagated to the rest of the function body that is specialized using this property. If the compiler supports other types and the function is called with a value of another type such as a flonum, another specialized version of the function body will be generated after discovering the type of the argument.

2.10. Optimizations

We explained that the compiler can add and remove properties from the compilation context and propagate this context to use it to specialize the code. The way the compiler internally represents a context is important because, depending on the representation it uses, some optimizations can easily be implemented, without requiring significant development effort.

When using a stack of types, types can easily be added when discovered, used to optimize the generated code and removed when consumed. To implement aliasing, the compiler can use an environment in the context that maps local identifiers to slots of the context's stack. Here is an example of a context representing a frame:

```
stack: (fixnum bool string)
env:   (  #f     y      x   )
```

In this context, there are three slots in the current frame. The local variable `x` that is a string, the local variable `y` that is a boolean, and a third value that is a temporary variable of type fixnum.

A notable feature of our approach is that compilation decisions are taken locally. Because each node is compiled independently (i.e. without knowledge of the control flow), optimizations with large scope cannot be easily applied. However, using a lazy design, code specialization and a representation of the compilation process with continuations, allows applying optimizations locally and propagating their effects to the rest of the compilation, resulting in optimizations

with a larger scope. We explain in Chapter 3 how we extend the technique to propagate the effects of these optimizations interprocedurally.

In this section, we explain that representing the compilation context using a stack allows applying these optimizations with low development effort.

2.10.1. Stack Mapping

An advantage of using a stack to represent values of the current frame is that this stack can easily be mapped to the machine execution stack. When the compiler visits a node representing a value (constant and variable nodes), its type is added to the context, the next slot on the execution stack is allocated for this value and code is generated (e.g. using the machine *push* instruction). When the compiler visits a node consuming values (e.g. arithmetic operator nodes), the type of the operands are removed from the context and code is generated to retrieve the operands from the execution stack.

2.10.1.1. *Example*

Figure 2.10 shows an example of lazy compiling the expression `(f64vector-ref v 2)`. `f64vector` is an homogeneous vector of double precision flonums. The `f64vector-ref` function reads the flonum at a specific index of the given `f64vector`. This example assumes that the compiler does not check f64vector bounds for this function. The left side of the figure shows the generated code and the right side shows the context propagated by the compiler and used to specialize the code. In this example, we assume that the compiler has discovered that `v` is a `f64vector` that is the only value in the frame and is located at `[fs-1]` (the first value from the bottom of the stack). Before visiting the nodes, no code is generated for this expression. Using the design presented in this chapter, the operand nodes are visited before visiting the operator. When the node of the first operand `v` is visited, code is generated to push its value to the execution stack using the location available in the context. The context is updated, the new slot contains a `f64vector` and is an additional copy of `v`. When the node of the second operand is visited, the code to push the constant `2` to the execution stack is generated, the context is updated with a new slot and the type `fixnum`. Note that the fixnum is encoded before being pushed to properly implement dynamic typing (its value and its type are both added to the stack in a single machine word). Finally, when the node of the operator is visited, the compiler uses the properties available in the context to generate

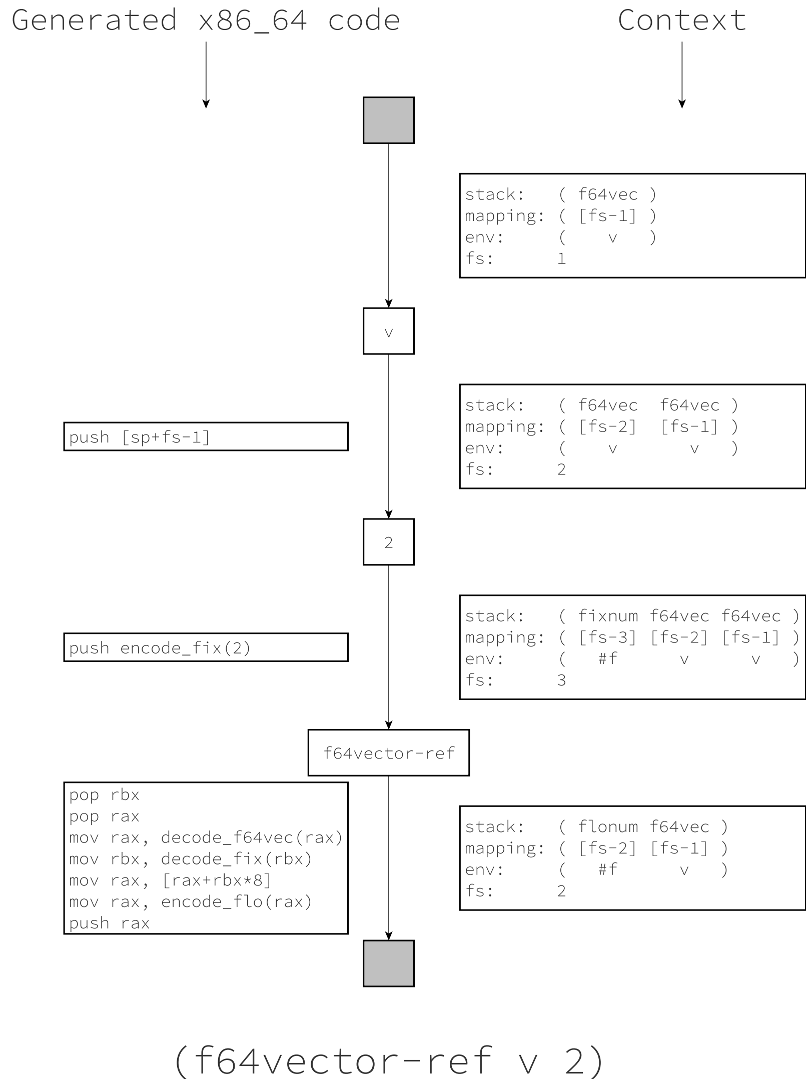


Fig. 2.10. Lazy compilation of the expression `(f64vector-ref v 2)` with generated x86_64 code (Intel syntax).

code with no type test, and to retrieve the locations of the operands in the execution stack. An additional instruction (`decode_f64vec` or `decode_fix`) is generated for each operand to extract its value from the machine word containing both its value and its type.

The operands are removed from the context, and because the compiler knows that `v` is a `f64vector`, it knows that the fetched value is a flonum and adds its location and type to the context. Because the type is propagated, no type test is required for the following instructions using this value. An additional instruction is inserted to encode the fetched flonum.

2.10.2. Register Allocation

A compiler uses register allocation to use registers to store the values of variables. The number of variables may be greater than the number of machine registers, but registers are much faster to access than memory. The goal of register allocation is to decrease the number of moves between registers and between registers and memory.

Register allocation can be done by coloring the interference graph [**31, 30, 39, 27, 127**]. Graph coloring allows generating efficient code but is not adapted to JIT compilation because compilation time is an important concern [**75, 29**]. Greedy techniques such as *linear scan* [**107**] that are not based on graph representations and more adapted to JIT compilation have been developed. However, linear scan depends on liveness analysis which is *(i)* computed by a backward static analysis thus not aligned with the goals of our project and *(ii)* difficult to apply if no intermediate representation is used. Here we explain how the compilation design allows easily implementing a simple greedy register allocator.

Using a stack in the context allows easily adding a greedy register allocation. When the compiler adds a type to the context, it allocates a free register to this slot instead of the next slot of the execution stack (in memory).

It is possible that several slots in the context's stack are allocated to the same location. For example, if a variable is used multiple times as an operand, it is added multiple times to the context's stack. The same register can then be used, meaning that no code is generated to add this variable.

If no free register is available when adding a type to the context, a slot of the context's stack associated with a register must be spilled to free a register. Any spilling heuristic can be used in this situation. In our implementation, we spill the oldest slot of the stack allocated to a register (i.e. the lowest slot of the context's stack).

In addition, using a stack in the context allows the compiler to automatically free the registers allocated to temporary values without requiring liveness analysis. When a type is removed from the context's stack because the value is consumed by an operation, its associated register is freed if it is not used by other slots in the stack.

The compiler can also use register allocation information to specialize the code. If a join point is reached multiple times with contexts in which register allocation information is different, the join point can be specialized for each context to avoid using additional moves

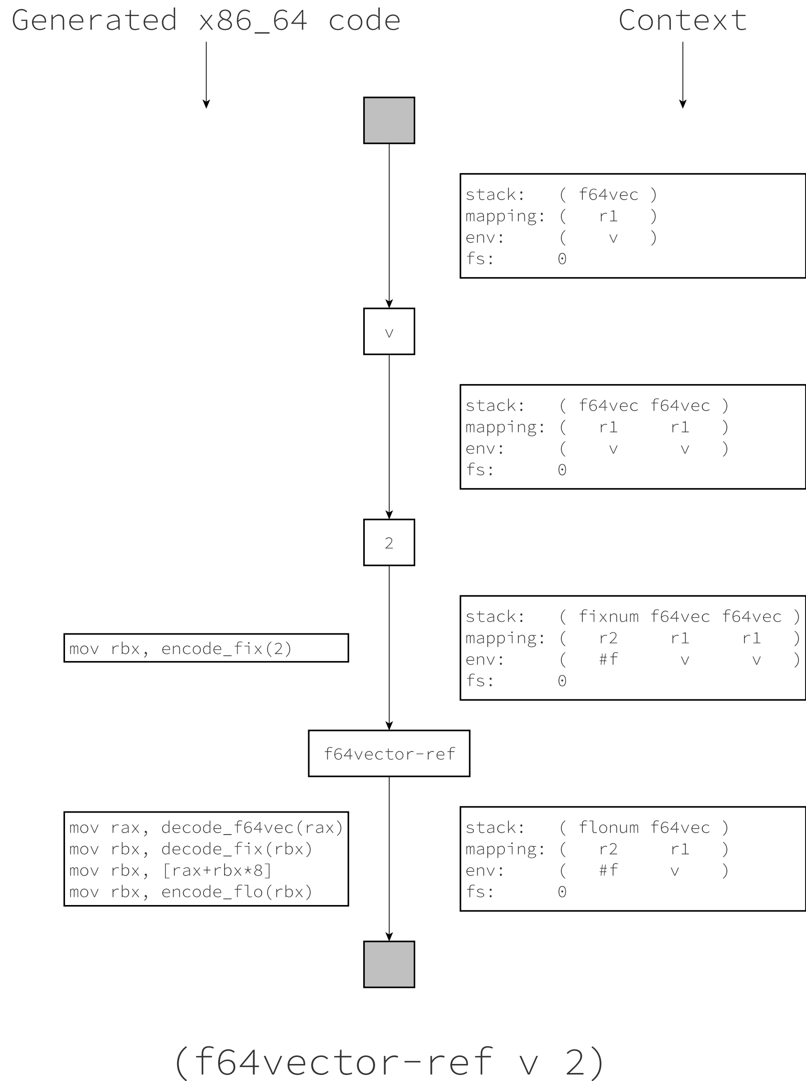


Fig. 2.11. Lazy compilation of the expression `(f64vector-ref v 2)` using lazy register allocation with generated x86_64 code (Intel syntax).

between registers to merge the contexts. This specialization causes an implicit tail duplication at the join point.

2.10.2.1. *Example*

Figure 2.11 shows an example of lazy compiling the expression `(f64vector-ref v 2)` with greedy register allocation. This example assumes that the compiler knows that `v` is a `f64vector` that is stored in the register `r1`.

When the node representing the first operand is visited, the context is updated to add the type. The environment is also updated because the added slot is a new location of `v`. Because the value is `v` and `v` is already allocated to a register, this register is used for this new slot in the context. No code is generated for this node.

When the node of the second operand is visited, the next available register is allocated to this slot (register `r2`). The context is updated with the type `fixnum` and register `r2`. The code to load the encoded constant `2` to its associated register is generated.

Finally, when the operator is visited, the compiler uses the properties available in the context to generate code with no type test and to use the registers allocated to the operands. The operands are then removed from the context. Because the operands are removed, and because it is not used by other slots in the stack, register `r2` is added to the set of free registers. Then, when a register needs to be allocated for the result of the operation, register `r2` is used.

We can see that adding a greedy register allocation to the context allows generating more optimized code with low effort.

2.10.3. Constant Propagation

Other optimizations that can easily be applied using this context representation are *constant propagation* and *constant folding*. When a node representing a constant is visited, the constant is added to the context's stack in addition to its type without requiring it to be allocated to a register. No code is generated to load this constant and the constant is propagated through the context to the following nodes.

When an operation is visited, the compiler benefits from the propagation of constants to generate more efficient code. If all the operands are constants in the context, and if the operation is side-effect free, the compiler can compute the result of the operation at compile time using the constants and add it to the context's stack. This is constant folding.

2.10.3.1. *Example*

Figure 2.12 shows an example of lazy compiling the expression `(f64vector-ref v 2)` using constant propagation. In this example, we assume the compiler knows that `v` is a `f64vector` that is stored in the register `r1`.

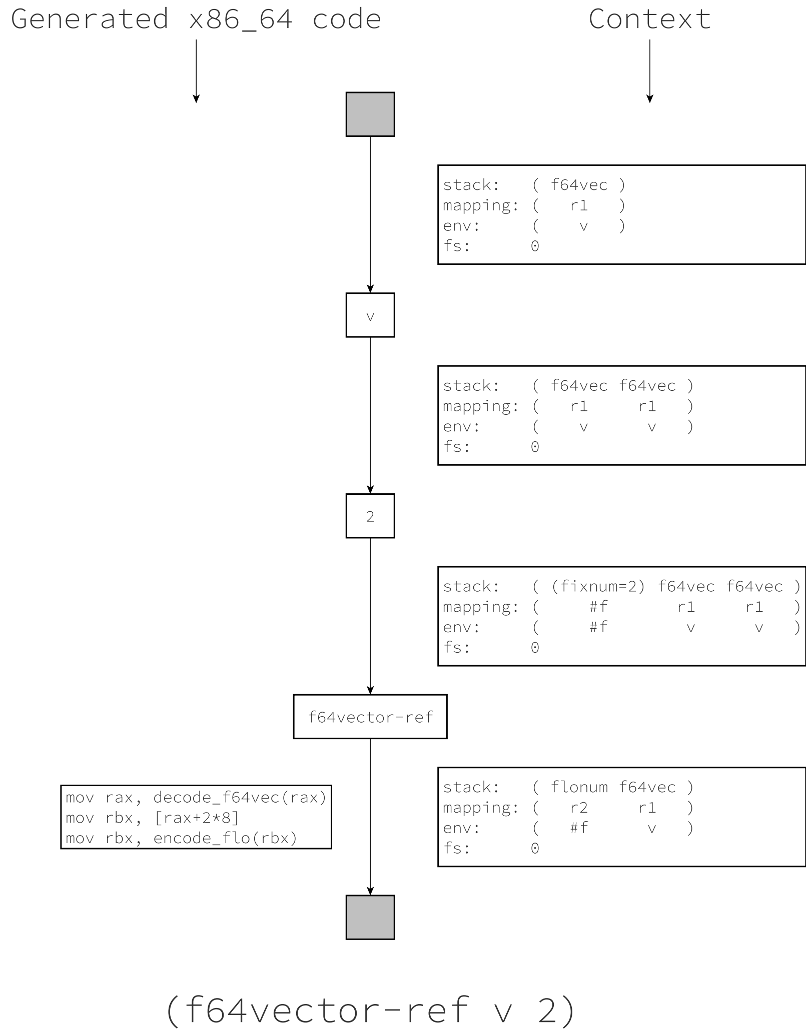


Fig. 2.12. Lazy compilation of the expression `(f64vector-ref v 2)` using constant propagation with generated x86_64 code (Intel syntax).

As in the previous example, when the node for `v` is visited, no code is generated and the context and the environment are updated.

The node of the second operand represents the constant `2`. When this node is visited, the type of the constant and the constant itself are added to the context's stack. Because it is a constant, no register is allocated to this slot and no code is generated.

Finally, when the node representing the operation is visited, the compiler uses the properties available in the context to generate code with no type check, using the registers allocated to the operands and with constant operands.

Once the code of the operation is generated, the context is updated with the operation result properties.

2.10.4. Value Unboxing

When the compiler discovers the type of a variable, it can safely handle it unboxed (in its decoded form) while its type is known. When an operation is visited, the compiler then knows that if the types of the operands are known, they already are unboxed. As soon as the type is lost (e.g. the limit in the number of versions is reached and the compiler falls back to a generic version), the value must be boxed. In Chapter 4, we explain in more detail how this optimization can be applied for all types with low effort by exploiting the value representation.

2.10.4.1. *Example*

Figure 2.13 shows an example of lazy compiling the expression `(f64vector-ref v 2)` using value unboxing. In this example, we assume the compiler knows that `v` is a `f64vector` that is stored in the register `r1`.

Because the type of `v` is known it is already unboxed in `r1`. When the operands are visited, no code is generated and the context is updated with the operand properties. When the operation node is visited, no unboxing code is generated for the operands. The `f64vector` operand is already unboxed in `r1` and the index is a constant. Because the compiler knows that the `f64vector-ref` function returns a flonum, the type of the result is known and propagated thus there is no need to encode it. An additional optimization is that, because the compiler knows that the result is a flonum, it can extend the register allocation and store this value in a specialized register if available in the target machine (e.g. the xmm registers on x86_64 machines). This value will then be handled unboxed and no register moves are needed for future uses of this value. In this example, the result is allocated to the flonum specialized `fr1` register.

We can see that for this example, using simple local optimizations based on the properties propagated through the context allows generating code for the `f64vector-ref` operation using a single x86_64 instruction, without requiring the use of static analysis to propagate constants and remove unboxing operations and without requiring liveness analysis to allocate registers and handle temporary values.

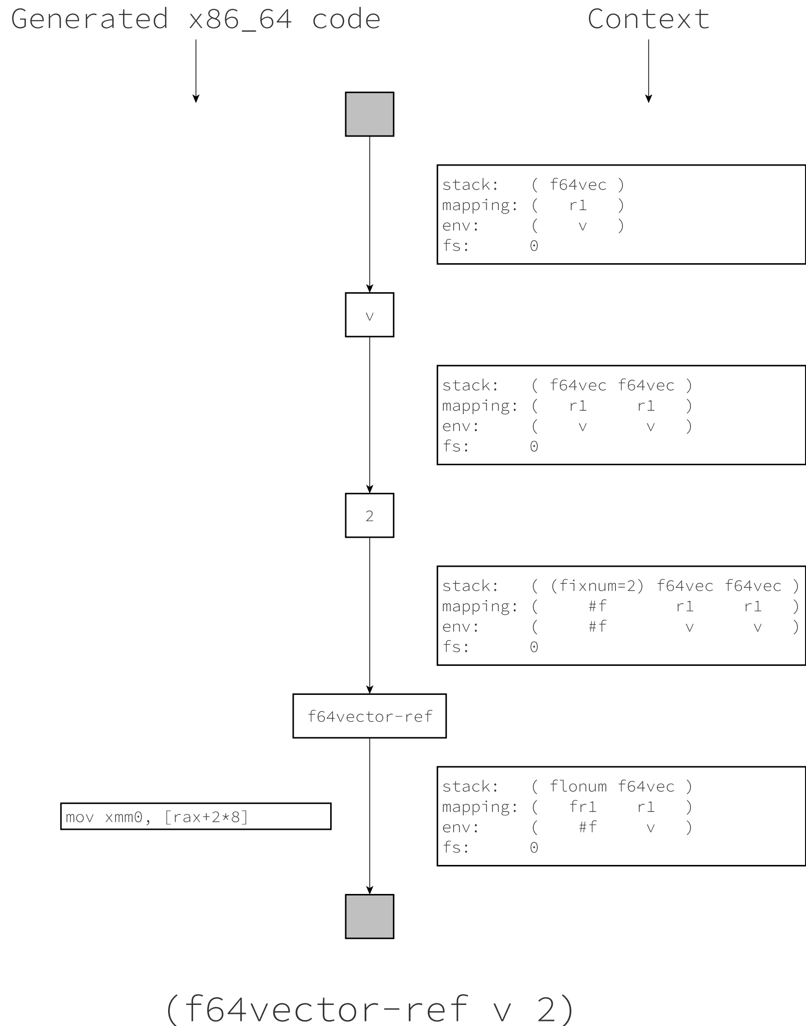


Fig. 2.13. Lazy compilation of the expression `(f64vector-ref v 2)` using value unboxing with generated x86_64 code (Intel syntax).

2.11. Summary

Generating code for a program represented by an AST is easy using a simple tree walking compiler. However, visiting and generating code AOT for each node does not allow delaying compilation which is required to implement dynamic techniques such as BBV. We explained how tree walking compilers can be extended to represent the compilation process using CPS. This representation allows, without much effort, to add to the compiler the ability to lazily generate code. We then explained how the tree walking compiler can be extended from this CPS representation to allow specializing and duplicating the code generated for the

AST nodes. Using these extensions, compilers can use BBV to specialize the generated code according to the dynamic properties of the executed program and generate code that is considerably more efficient than the code that is generated by a naive AOT compilation of the AST. This allows applying classical local optimizations as well as some optimizations with a larger scope used by state-of-the-art JIT compilers for dynamic languages while keeping the development effort low.

Chapter 3

Interprocedural Code Specialization in the Presence of Higher-Order Functions

Part of the work presented in this chapter is published in the two papers:

- Baptiste Saleil and Marc Feeley. Type Check Removal Using Lazy Interprocedural Code Versioning. In *Scheme and Functional Programming Workshop, 2015.* [**113**]
- Baptiste Saleil and Marc Feeley. Interprocedural Specialization of Higher-Order Dynamic Languages without Static Analysis. In *31st European Conference on Object-Oriented Programming, ECOOP, 2017.* [**114**]

In this chapter, we expand on the ideas presented in those papers.

3.1. Introduction

In its original definition, BBV is strictly intraprocedural. This means that if the compiler is able to accumulate properties when compiling a function *f*, these properties are used to specialize the code of *f* only, they are not propagated through function calls and returns. These properties have to be discovered again in the functions and the continuations invoked in *f*, leading to inefficiencies. Note that unlike in Chapter 2, *continuation* refers to call continuations and not to compilation continuations in this chapter.

3.1.1. Function Calls

When the compiler generates code for a call site, it is possible that it discovered and propagated dynamic properties of the program and that it used them to specialize the code of this site. To generate better code, the compiler could generate a specialized version of the callee function using the properties available at the call site.

```
 1: (define make-sumer
 2:   (lambda (n)
 3:     (letrec ((f (lambda (x)
 4:                   (if (> x n)
 5:                       0
 6:                       (+ x (f (+ x 1)))))))
 7:       f)))
 8:
 9: (define sum-to-10 (make-sumer 10))
10: (define sum-to-pi (make-sumer 3.14))
11:
12: (println (sum-to-10 6))    ; 6 + 7 + 8 + 9 + 10
13: (println (sum-to-10 7.5))  ; 7.5 + 8.5 + 9.5
14: (println (sum-to-pi 1.10)) ; 1.10 + 2.10 + 3.10
```

Fig. 3.1. Scheme code of an arithmetic sequence generator with a common step of 1.

Figure 3.1 shows a program where more efficient code can be generated by BBV when type properties are propagated interprocedurally. The Scheme function `make-sumer` generates bounded arithmetic sequence calculators with a common step of 1. Two sumers are created using respectively an upper bound of 10 (a fixnum) and 3.14 (a flonum). When intraprocedural BBV is used when executing this code, the `>` operator allows the compiler to discover the type of the argument `x` in the expression `(> x n)`. The implicit type dispatch conceptually tests for fixnum and flonum types for `x` (and `n`). The discovered type of `x` is propagated and the rest of the body of the function `f` is specialized for the type of `x`. When the compiler generates the recursive call to the function, it knows the type of the argument but in the absence of interprocedural specialization, the compiler generates the call to a single generic version of the function. Using interprocedural code specialization, the compiler generates a call to another version of the function `f`, specialized for the type of its argument. This specialized version of `f` does not even need to test the type of `x` in the expression `(> x n)`. The type of `x` is discovered in the first call to `f` and never tested again in the subsequent recursive calls.

This means that using interprocedural BBV, a function now possibly has several entry points, each specialized for the properties known when generating each site where this function is called. A specialized function entry point is then uniquely identified by:

$$\mathcal{I}_{\text{lambda}}\ ,\ \mathcal{P}_{\text{call}}$$

with $\mathcal{I}_{\text{lambda}}$ being the identity of the function in the source code and $\mathcal{P}_{\text{call}}$ the set of properties known at the call site used for specialization (e.g. the type of the arguments).

Generating a call to a specialized function entry point when generating a call site is relatively easy when the identity of the callee function is known. The compiler can check in the code cache of $\mathcal{I}_{\text{lambda}}$ if a version has already been specialized for the current $\mathcal{P}_{\text{call}}$. If it is the case, it generates a call to this version. If it is not the case, it creates a compilation stub that, when triggered, generates the version and patches the call site to allow jumping directly to this version in subsequent executions of this call.

However, with higher-order languages offering first-class functions, the identity of callee functions is not generally known at compile time. Moreover, it is possible that several functions are called at the same call site. In this general situation, the compiler knows $\mathcal{P}_{\text{call}}$ but does not know $\mathcal{I}_{\text{lambda}}$. It must insert a dynamic dispatch at call sites to dispatch the execution to the entry point specialized for $\mathcal{I}_{\text{lambda}}$, $\mathcal{P}_{\text{call}}$.

3.1.2. Captured Properties

In addition to the properties known at call sites, a function can be specialized using the properties known when creating its lexical closure. In this case, a specialized function entry point is uniquely identified by:

$$\mathcal{I}_{\text{lambda}} \text{ , } \mathcal{P}_{\text{call}} \text{ , } \mathcal{P}_{\text{closure}}$$

with $\mathcal{P}_{\text{closure}}$ the properties known when creating the lexical closure (e.g. the type of the free variables).

For example, in Figure 3.1, two closure instances of the function `f` are created. If the compiler uses interprocedural BBV as presented above, the type of the constant 10 is propagated through the call `(make-sumer 10)` meaning that the type of the captured variable `n` (fixnum) is known when creating the first instance of the closure of `f`. In the same way, the type of the captured variable `n` (flonum) is known when creating the second instance of the closure of `f` by the call `(make-sumer 3.14)`. If the compiler is able to use this property to specialize the code when generating the function `f`, then no type test is needed for `n`.

3.1.3. Function Returns

An analogous problem exists for function returns. The properties known at function returns can be propagated to the continuations. A continuation then possibly has several

entry points, each specialized for a set of properties. A call continuation can be seen as a closure representing a function that is called with a single argument being the returned value and capturing the local variables available at the call site where it is created. In this case, $\mathcal{I}_{\text{lambda}}$ is the identity of the continuation, $\mathcal{P}_{\text{call}}$ is the set of properties known at the function return and used for specialization and $\mathcal{P}_{\text{closure}}$ is the set of properties known when creating the object representing the continuation. In the case of typing, $\mathcal{P}_{\text{call}}$ is the type of the returned value and $\mathcal{P}_{\text{closure}}$ represents the type of the local variables captured when creating the object representing the continuation.

In the following section, we explain how we extend the classical flat closure representation to store multiple specialized entry points. We then explain how this representation can also be used for continuations with multiple entry points. We then present a dynamic dispatch in constant time using this closure representation. Our approach is *position invariant*, allowing the generation of calls to the specialized entry points without knowing $\mathcal{I}_{\text{lambda}}$.

3.2. Related Work

3.2.1. Function Entry Points

3.2.1.1. *Inline Expansion*

Inline expansion [**41**] is a form of interprocedural code specialization. Function inlining is used to decrease the overhead associated with calling and returning from a function by instantiating the body of callee functions at call sites. Instantiating the body allows using the properties known from the caller to optimize the body of the callee, enabling additional optimizations such as constant propagation from the caller and Common Subexpression Elimination (CSE) using the properties known both from the caller and the callee to name a few. Inlining is a well-known optimization used by most compilers for various languages [**32, 102, 105**] including Scheme compilers [**121, 74, 47, 54, 18**].

However, inlining for functional and object-oriented programming languages is problematic because the identity of the callee is not generally known at call sites.

Techniques such as Inline Caching [**44**] and Polymorphic Inline Caching (PIC) [**66**] allow inlining to be applied for polymorphic sites at additional run time costs. Using PIC, a linear search is executed at each polymorphic call to dispatch the execution.

Because it is position invariant, our technique allows using interprocedural specialization in the presence of higher-order functions and polymorphic and megamorphic calls are handled without additional cost.

3.2.1.2. *Static Analyses*

Static analyses can be used to collect properties to interprocedurally specialize the generated code. For example, the *k*-CFA [**125, 126**] analyses use abstract interpretation to statically attach types to expressions. Static analyses can also be used to infer the identity of callee functions used at call sites, addressing the issue we presented in the previous section.

However, these analyses have high complexity (e.g. $\mathcal{O}(n^3)$ in the worst case for *0*-CFA [**120, 94, 93**]) making them unsuitable for use in JIT compilation because the compilation time impacts the overall execution time. In addition, these analyses are imprecise, the identity of the callees cannot generally be determined for all sites. Because they are conservative, the inferred properties can also include properties that are not used at execution.

More recently, new static analyses for type inference allowing specializing code and more adapted to JIT compilation have been discovered [**87, 64**]. However, such analyses are limited because they generally assign a single type to the values. The generated code then cannot benefit from code duplication to specialize polymorphic code.

3.2.1.3. *Code Customization*

Function duplication for dynamic languages has first been studied at the end of the 80's and the beginning of the 90's with the Self programming language [**137, 33**]. Self is a pure object-oriented language offering dynamically-bound messages. Self uses function specialization through techniques such as Customization and Splitting [**32**]. Customization allows compilers to duplicate functions according to the characteristics of call sites. In the implementation presented in [**32**], functions are customized according to the dynamic type of the receivers used at call sites. A test is then inserted in function prologues [**69**] to determine the type of the receiver and to branch to the customized version. This test implements the dynamic dispatch required to use interprocedural specialization in the presence of higher-order functions or virtual methods.

As presented in [**32**], Customization only uses the types to specialize the code and the type of the receiver only is used to specialize code. However, they observed that *method arguments are one of the largest sources of unknown type information* [**32**] in their system.

They briefly discuss a solution consisting of using the type of the arguments to customize the code and dispatch the execution in the prologue of the methods at execution time.

Our solution is instead based on a dispatch at call sites. Because our dispatch is position invariant, it can be executed in constant time with less impact on performance.

3.2.1.4. *Basic Block Versioning*

Chevalier-Boisvert and Feeley [**38**] presented a technique based on BBV allowing inteprocedural specialization. In this technique, the identity of functions is added to the context used by the compiler to specialize the code. When the compiler generates a call site, it is then possible that this site is specialized for the identity of the callee. In this situation, because the identity is known at compile time, the compiler generates a jump to the specialized version associated with the current calling context, or generates a stub ready to generate it and patch the call site if it does not exist. If the identity of the callee is unknown because the compiler has not been able to propagate it to this site, a call to a generic non-specialized entry point is generated.

Using this technique, each polymorphic call site is specialized, and duplicated, for all the functions used as callee for this site. The technique may then cause a code-size increase to handle polymorphic and megamorphic sites. In addition, the technique is limited because if BBV is not able to propagate the callee identity, the function cannot be specialized.

Our dispatch instead allows using interprocedural specialization with no increase in the code size for polymorphic and megamorphic call sites because there is no need to specialize the code using function identities. In addition, the technique can be used even when the compiler does not know the identity of callee functions. It is then not limited by the amount of properties BBV is able to propagate.

In an execution environment based on BBV, it is possible to combine both techniques. The compiler can propagate function identities to easily jump to specialized entry points and use our approach as a fallback to handle cases in which the callee identity is unknown.

3.2.1.5. *Tracing*

Tracing JIT compilation has been used to efficiently implement dynamic languages [**56, 95, 18**]. Because the recording phase of tracing follows function calls, the function and continuation bodies recorded during that phase can be inlined in the traces. Generated code can then be specialized for each trace.

In addition to the differences we mentioned in Section 2.2.1, tracing compilation is based on identities contrary to our approach. The recording phase of a trace speculates that the path will not change for subsequent executions of traces. Guards are then inserted in the generated code to validate the speculations [**56**].

3.2.2. Call Continuations

There has been less research work on specialization of call continuations. We have presented the issue of implementing function specialization for functional and object-oriented languages. Because the identity of the callee function is not generally known at call sites, it is problematic to branch to the specialized version. For many programming paradigms, including functional, object-oriented and imperative programming, the same issue exists when implementing call continuation specialization. When generating code for a function f, because f can be called from various caller functions, the compiler does not usually know the identity of the continuation when generating code for f's return. In addition, most of function returns are polymorphic (i.e. more than one continuation are invoked at a given function return) making continuation specialization difficult to implement efficiently.

3.2.2.1. *Continuation-Passing Style*

A solution to specialize continuations is to transform the executed program to CPS, a technique that is widely used by compilers for functional languages, including Scheme [**130, 81, 11, 9**]. If a program is transformed to CPS, control between functions is explicitly passed using functions representing call continuations. After the program is transformed, function returns are represented by calls to these continuations (i.e. by function calls). Because function returns are now represented as function calls, all the techniques presented in the previous section, used to specialize functions, can be used to specialize continuations.

3.2.2.2. *Basic Block Versioning*

Chevalier-Boisvert and Feeley [**38**] presented a speculative technique based on BBV allowing a limited form of continuation specialization with no run time cost.

Using this technique, the compiler specializes the continuation of a call to a function f only if:

(1) The identity of f is known when generating this call

(2) f always returns values of a single type

If the compiler discovers that a value of at least a second type is return by f when generating a specialized return, the previously specialized continuations of calls to f are invalidated, deoptimized and a generic fallback version is used instead.

These conditions represent an important issue to specialize call continuations for Scheme and other dynamic languages. A simple example is functions returning lists. Functions processing lists return values of (at least) two types, *null* and *pair*. If the compiler uses this approach, continuations of calls to these functions will not be specialized.

Our technique instead allows continuation specialization without these limitations. The compiler is able to specialize call continuations even when the function returns values of different types. In addition, our technique does not cause deoptimization. However, an extra indirection is needed and executed at call sites.

3.3. Flat Closure Representation

A closure [**84**] is a data structure used to implement first-class functions in a language using lexical scoping. It is an object representing a function and contains two elements:

(1) The address of the code of its function. At call sites, the compiler generates code to retrieve this address from the closure object and to jump to it to implement the call. With JIT compilation, the compiler first writes the address of the stub of the function while it is not yet generated. When the function is generated, the address of the stub is replaced by the address of the function.

(2) The variables of the environment captured when creating the closure object (i.e. the *free variables*). A free variable is a variable that is used by the function and declared in an enclosing scope. These values must be captured when the closure is created.

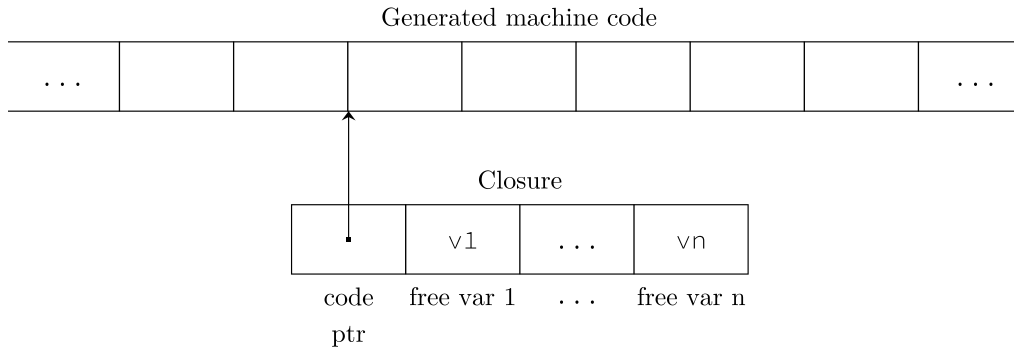


Fig. 3.2. Flat closure representation.

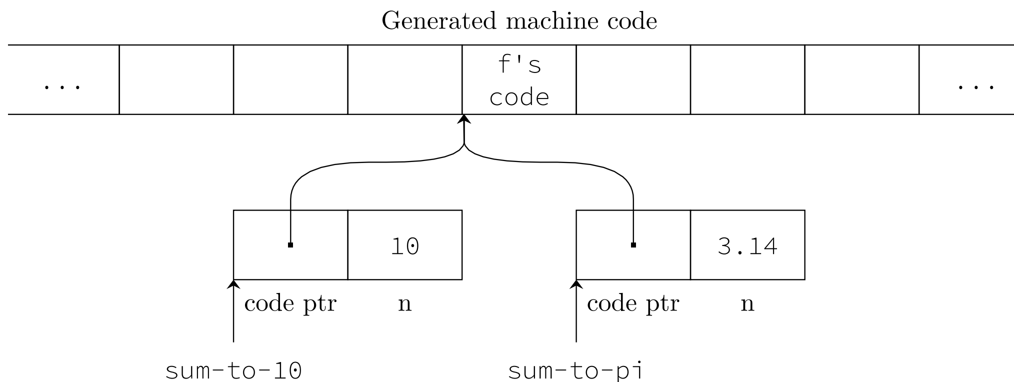


Fig. 3.3. Flat closures created for the function `f` of the example in Figure 3.1.

There are several approaches to represent the environment [**10**]. Each technique has advantages and disadvantages but the flat closure representation is one of the simpler and most used techniques, including by Scheme compilers [**121, 61, 47**].

With the flat closure representation, the free variables are copied in the closure object next to the code pointer. Figure 3.2 shows how flat closures are typically represented by compilers.

This more or less abstract representation is used in the examples in the rest of this chapter but variations in the details are possible.

3.3.1. Example

In the code we presented in Figure 3.1, two closures are created for the function `f`. The first closure is created with the call `(make-sumer 10)` and kept in `sum-to-10`. When the `make-sumer` function is executed, the free variables of `f` are captured to create its closure. The closure object is then returned. This object contains the address of `f`'s code and the

value of its single free variable `n` which is `10`. The second closure is created with the call `(make-sumer 3.14)` and kept in `sum-to-pi`. This closure contains the address of `f`'s code and the value of the free variable `n` which is `3.14`. Figure 3.3 shows the representation of the two closures `sum-to-10` and `sum-to-pi` after the code of the example is executed.

3.4. Flat Closure Representation Extension

In this section, we explain how we extend flat closure representation to allow the compiler to specialize functions using $\mathcal{P}_{\text{call}}$. We explain how the compiler can specialize functions using $\mathcal{P}_{\text{closure}}$ in Section 3.6. Here $\mathcal{P}_{\text{closure}}$ is ignored.

When using interprocedural code specialization, functions may have several entry points, each specialized for a specific $\mathcal{I}_{\text{lambda}}$, $\mathcal{P}_{\text{call}}$. However, because a closure has a single code pointer, it is not possible to store all the specialized entry points in the closures without extending their representation. Instead of storing a single code pointer, the compiler could store all the entry points before the free variables. However, the compiler does not generally know the exact set of specialized entry points the function will have when a closure is created. Consequently, it does not know the number of slots it must allocate to store the entry points. Our approach is to instead use an external table, the entry point table, to store the entry points of the function and to store the address of this table in the closure. All the closure instances of the same $\mathcal{I}_{\text{lambda}}$ share the same entry point table. When a new entry point is generated for a function, it is stored in the next available slot of its table.

Figure 3.4 shows how the flat closure representation is extended to use an external entry point table. Each slot in the table is associated with a context that has been used to generate a specialized version of the function. If the compiler knows $\mathcal{I}_{\text{lambda}}$ when generating a call site, it generates the code to retrieve the address of the version from the slot associated with the current $\mathcal{P}_{\text{call}}$ and uses it for the call. However, because the compiler does not generally know $\mathcal{I}_{\text{lambda}}$, it does not know at compile time which slot in the entry point table of the closure used at this site is associated with the current $\mathcal{P}_{\text{call}}$. It does not even know if this slot exists or if it must be generated. To address this issue, a dynamic dispatch must be inserted in the code generated for the call.

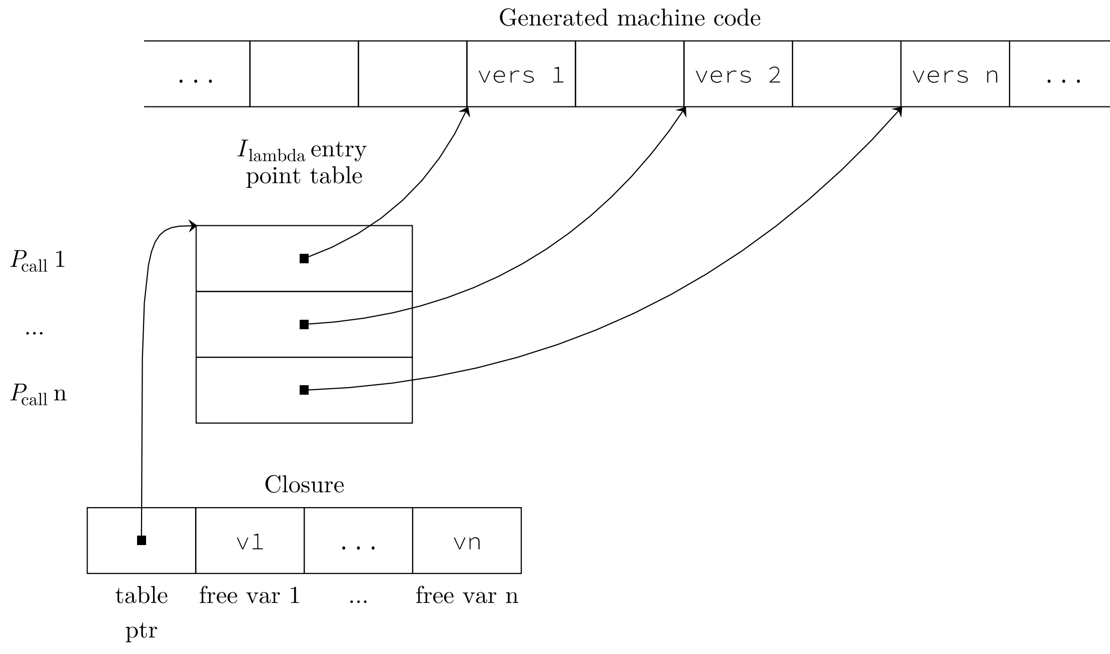


Fig. 3.4. Extended flat closure representation with an entry point table.

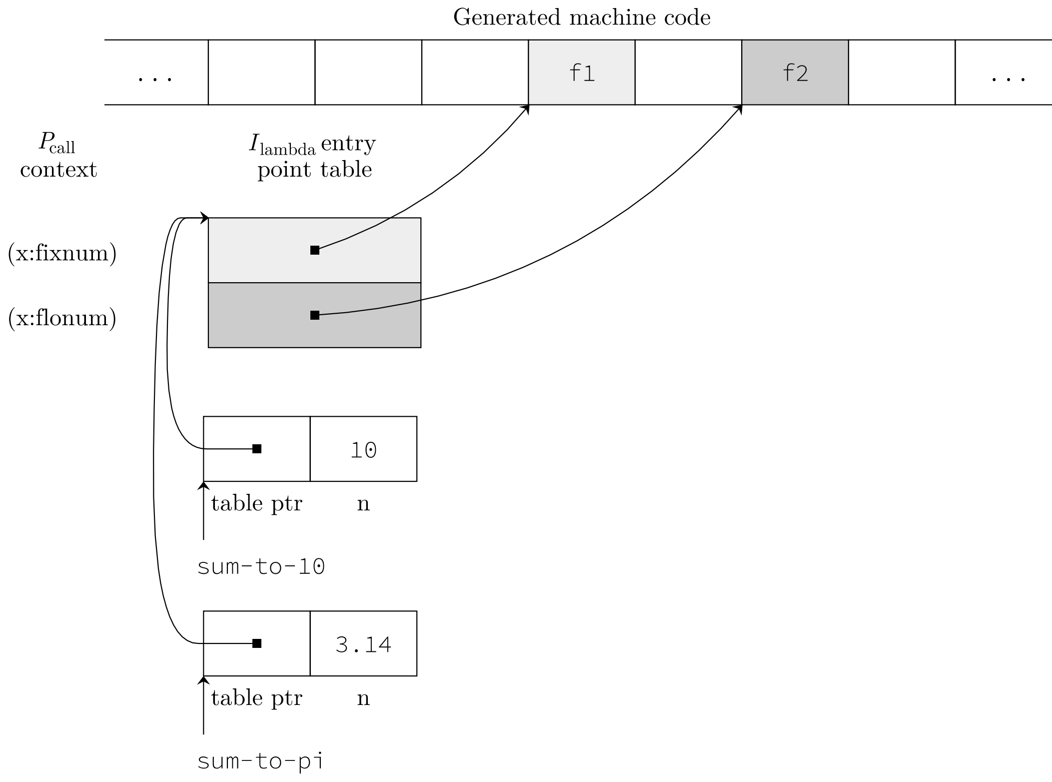


Fig. 3.5. Closures using entry point tables created for the function `f` of the example in Figure 3.1.

3.4.1. Example

Assuming that the compiler uses this extension and knows the identity of the callee functions at call sites, Figure 3.5 shows the state of the two closures `sum-to-10` and `sum-to-pi` after the code of the example is executed. The two closures share the same entry point table. The table contains two entry points. The first is specialized for `x` being a fixnum for the call `(sum-to-10 6)` and the second is specialized for `x` being a flonum for the call `(sum-to-10 7.5)`. Because a version has already been specialized for `x` being a flonum, no other version is generated for the call `(sum-to-pi 1.10)`, the compiler reuses the second version for this call.

3.5. Dynamic Dispatch

The dispatch can be implemented in various ways. A solution is to use PIC at call sites. A PIC would allow jumping to the entry point corresponding to the current $\mathcal{P}_{\text{call}}$ depending on the $\mathcal{I}_{\text{lambda}}$ used at execution. When a new function is used at this call site (i.e. the PIC fails, a new $\mathcal{I}_{\text{lambda}}$ is observed at execution), the compiler adds this function to the PIC for future executions of this site with this $\mathcal{I}_{\text{lambda}}$. Another solution is to use hashing. Using hashing, the compiler can retrieve the entry point by hashing the identity of the function from the closure at call sites. However, these techniques significantly impact the execution time, in particular for polymorphic call sites. In the case of PIC, the more $\mathcal{I}_{\text{lambda}}$ are observed at a given call site, the more the run time cost increases for this site. In the case of hashing, the identity must be hashed each time the call is executed, degrading performance.

Our approach is instead based on position invariance of the entry points in the tables making the implementation similar to virtual function tables [**42**] that are popular to implement message dispatch for C++ [**48**]. The run time cost of our approach in then similar to the cost of a virtual method call using virtual function tables [**46, 100**].

3.5.1. Global layout

To maintain position invariance of entry points, all the entry point tables share the same layout that we call the *global layout* regardless of their associated $\mathcal{I}_{\text{lambda}}$. The compiler associates a $\mathcal{P}_{\text{call}}$ to each slot of the layout. In a JIT compilation context, the global layout is initially empty (i.e. there is no association) and new associations are added on-the-fly. When generating a call site, if the compiler observes a $\mathcal{P}_{\text{call}}$ that has never been used before, it

associates the next available slot in the global layout to this $\mathcal{P}_{\text{call}}$. The global layout has a size equal to the total number of unique $\mathcal{P}_{\text{call}}$ observed during the execution of the program. If applied to typing, the global layout has a size equal to the total number of argument type combinations observed at call sites during the execution of the program.

3.5.2. Implementation

When the compiler creates an entry point table, it does not know the final size of the global layout meaning that it does not know the final size of the entry point table it is currently creating. To address this issue, one of these strategies can be used:

(1) The tables are created with a fixed size (S). One of the slots of each table is initially associated with a fallback generic context. Each time a new $\mathcal{P}_{\text{call}}$ is used at a call site, the compiler allocates the next slot of the global layout to this $\mathcal{P}_{\text{call}}$. When all the slots are used, the compiler uses the fallback slot for all the subsequent contexts and stops specializing function entry points.

(2) The tables are initially created with a fixed size *S1*. When all the slots of the global layout are used, all the entry point tables are reallocated with a new larger size *S2* and copied to enable more slots. This strategy implies that the compiler patches all the live closures to update the table addresses, a process that is akin to the tracing done by a Garbage Collector (GC) and that can be integrated to it.

(3) The tables are created with a fixed size S. One of the slots of each table is reserved to handle the case in which the global layout is full. When the layout is full, all the entry point tables can be extended by creating new tables and chaining them using the reserved slots, adding an extra level of indirection for the new $\mathcal{P}_{\text{call}}$ cases.

Regardless of the strategy the compiler uses, a newly created entry point table is initially filled with the function stub address. While the function is not called, the stub is not triggered and no code is generated for the function. Note that the stub cannot be collected once an entry point is generated because it may be triggered to generate other entry points.

When the compiler generates the code for a call site, it first retrieves the index I of the slot associated with the current $\mathcal{P}_{\text{call}}$ in the global layout. It allocates a new index when this $\mathcal{P}_{\text{call}}$ is observed for the first time. The compiler then generates the code that retrieves the

```
1    push 0203FF90h  ; setup return address (initially stub address,
2                    ; patched with the return address later)
3    mov  rax,  1    ; setup the number of arguments
4    mov  rbx, 40    ; setup first argument
5    jmp  [rsi+7]    ; get code pointer from the closure and jump to it
```

Fig. 3.6. Example of x86_64 function call sequence using flat closures and intraprocedural only code specialization (Intel syntax).

```
1    push 0203FF90h     ; setup return address (initially stub address,
2                       ; patched with the return address later)
3    mov  rbx, 40       ; setup first argument
4    mov  rdx, [rsi+7]  ; get entry point table from the closure
5    jmp  [rdx+I*8]     ; jump to entry point at index I
```

Fig. 3.7. Example of x86_64 function call sequence using flat closures and interprocedural code specialization with entry point tables (Intel syntax).

entry point table from the closure and retrieves the address of the Ith entry point from the table and jumps to this address.

Figure 3.6 shows an example of a x86_64 code sequence typically generated by compilers to call a function represented by a flat closure without using interprocedural code specialization. The compiler first generates code to push the return address to the stack (that is initially a stub in this example). It then generates an instruction to write the number of arguments used by the call to a specific register (line `3`). The callee function uses this value to check if it is called with a valid number of arguments. The arguments are then moved to their associated registers (line `4`, in this case there is a single argument). Finally, the compiler generates a jump to the address of the entry point retrieved from the closure stored in the register `rsi` (line `5`).

Figure 3.7 shows an example of a x86_64 code sequence generated by a compiler using interprocedural specialization with entry point tables. The highlighted line shows the additional indirection necessary to implement the call using entry point tables. Before the

jump, the compiler inserts an instruction to first get the table from the closure stored in the register `rsi` (line `4`). It then generates the code to get the entry point from this table, using the index of the slot associated with the currently used context, and generates the jump (line `5`). When the call is executed, if this slot of this table is used for the first time, the stub that is initially written in the table is triggered and calls the compiler. We explain in Section 3.5.2.1 how the compiler retrieves the $\mathcal{P}_{\text{call}}$ it must use to specialize the code when a stub is triggered. Once the version is generated, the slot in the table is replaced by the version address for subsequent calls.

Note that the instruction used to pass the number of arguments to the callee function is not necessary. When a specialized entry point is generated, the compiler knows the $\mathcal{P}_{\text{call}}$ used at the call site. Because $\mathcal{P}_{\text{call}}$ contains the properties associated with each argument, it also contains the number of arguments. Consequently, a specialized entry point is specialized for a specific number of arguments and no check is needed in the callee function. When generating a specialized entry point, if the number of arguments in $\mathcal{P}_{\text{call}}$ is valid, the entry point is generated with no check and if the number is invalid, an error is raised.

Because all the entry point tables share the same layout, it is likely a table contains holes in the slots associated with $\mathcal{P}_{\text{calls}}$ used to generate specialized entry points of other functions but never observed for the function this table is associated with. These holes impact the memory footprint of the tables. However, because a table associated with a function is shared by all its closure instances, the compiler creates a number of tables equal to the number of functions observed during the execution of the program making this impact not significant. In Chapter 6, we study the memory footprint of the tables for Scheme programs.

3.5.2.1. *Retrieving the Contexts Used at Call Sites*

There are several ways to retrieve the $\mathcal{P}_{\text{call}}$ used at the call site when generating a specialized entry point of a function.

The first solution is to rely on the return address. When a stub representing a call continuation is created when generating a call site, the compiler stores in the stub the index of the slot, in the global layout, associated with the currently generated call. When a continuation stub is triggered, the compiler writes the index, stored in the stub, in the generated code before the code of the continuation. When a function stub is triggered to

```
1   push 0203FF90h     ; setup return address (initially stub address,
2                      ; patched with the return address later)
3   mov  rbx, 40       ; setup first argument
4   mov  r11, I        ; use context at index I of the global layout
5   mov  rdx, [rsi+7]  ; get entry point table from the closure
6   jmp  [rdx+I*8]     ; jump to entry point at index I
```

Fig. 3.8. Code of Figure 3.7 extended to pass the index associated with the $\mathcal{P}_{\text{call}}$ used at the call site to callee function stubs. When the call triggers a stub, the compiler retrieves the index from the register `r11` and retrieves $\mathcal{P}_{\text{call}}$ from this index. When the call does not trigger a stub, the index value is not used.

generate a specialized entry point, the compiler first gets the return address of this call from the execution stack. There are two possible situations:

(1) This return address represents a continuation stub. In this situation, this stub stores the index associated with its call site. The compiler retrieves it from this stub.

(2) This return address is the address of a generated continuation. In this situation, the compiler retrieves the index from the generated code (before the continuation code).

Another solution that may have a negative impact on performance but that is easy to implement is to directly pass the index used at the call site to the function stub as an argument. This solution is used in our implementation. In LC, when generating a call site, an additional instruction is generated to move the index associated with this call site to a specific register. When a call triggers a stub, the compiler retrieves the index from this register. The main disadvantage of this solution is that this additional instruction is executed at each call, even when the call does not trigger a stub. Figure 3.8 shows how the code in Figure 3.7 is extended to implement this approach.

Another solution that does not impact the performance of calls is to create a unique stub for each slot of each table instead of creating a single stub per table. Each stub stores the index of the global layout it is associated with. Consequently, when a stub is triggered, it already knows the $\mathcal{P}_{\text{call}}$ used for this call. The main disadvantage of this solution is that it

increases the amount of memory used for the stubs. However, unlike the previous solutions, once a stub is triggered and its associated entry point is generated, it can be collected.

3.5.3. Optimizations

The entry point tables allow generating more efficient code because they allow specializing functions. However, performance is also negatively impacted by the indirection added at call sites. To decrease this impact, several optimizations can be applied on the implementation of the entry point tables. In the rest of this section, we take type properties (the type of the arguments) as an example of property propagated and used to specialize the code.

3.5.3.1. *Slot Preallocation*

If the compiler uses fixed size tables, it stops specializing code when the global layout is full. However, it is possible that some contexts observed after the compiler decides to fallback to the generic entry point are extensively used in the program. This is for example the case if the executed program makes a lot of initialization work using many contexts. It is then possible that these contexts, specific to the initialization, fill the global layout.

To solve this issue, the compiler can preallocate some slots of the global layout. These slots are statically associated to contexts for which the compiler knows they allow generating efficient code. The compiler can preallocate a context if a static analysis determines that it is extensively used in the program. It can also preallocate the most used contexts. For example, the compiler can preallocate the n first slots of the global layout to the contexts representing a function call with a single argument with known type, for the n types the compiler supports. This ensures that all the calls to functions with a single argument with known type use a specialized entry point. The compiler can then allocate the remaining slots on-the-fly.

3.5.3.2. *Generic Entry Point*

Another issue is that, when generating a call site, when there is no known property in $\mathcal{P}_{\text{call}}$ (i.e. the compiler uses a generic context), the indirection used to retrieve the generic entry point from the table still needs to be generated at this site. In this situation, the code of the callee function is not specialized and performance is negatively impacted by the indirection. To address this issue, the compiler could, as in the original flat closure representation, store

the generic entry point directly in the closure, next to the address of the entry point table. The compiler can then use the generic entry point at no additional cost compared to the flat closure representation when no property is known in $\mathcal{P}_{\text{call}}$ or when it determines that there are not enough properties in $\mathcal{P}_{\text{call}}$ to use a specialized entry point. The drawback of this optimization is that, to store this entry point, the closures contain an additional slot impacting the overall amount of memory used by the program.

3.5.3.3. *Specific Entry Points*

In a similar way, the compiler can preallocate slots, each associated to a $\mathcal{P}_{\text{call}}$, in the closure next to the address of the entry point table to avoid the indirection when these specific $\mathcal{P}_{\text{call}}$ are used. For example, if an analysis used by the compiler determines that the argument type combination `(fixnum, flonum)` is extensively used by the program, the compiler can preallocate a slot for these argument types in the closure. Again, the drawback of using this optimization is that the overall amount of memory used by the program is negatively impacted.

3.5.3.4. *Heuristics*

The compiler can use heuristics to determine that a context does not offer enough optimization opportunities to be allocated to a slot. A typical example for Scheme is the use of a $\mathcal{P}_{\text{call}}$ representing a context with many arguments. Such contexts probably means that the callee function uses the rest parameter (i.e. it is a variadic function). In this situation, because the arguments are moved to a list, if the compiler does not track the type of compound data type, the type of the arguments will be lost. The compiler can then fall back to the generic entry point, use an existing entry point specialized for a most generic context or only keep the properties on the n first arguments. Table 3.1 shows the total number of function calls executed per number of arguments and per benchmark, suggesting that the compiler can use a small n to limit the size of the global layout.

	0	1	2	3	4	5	6	7	8	9	⩾10
ack			100.0								
array1				100.0							
boyer		3.2	96.8								
browse			34.5	28.2	33.9	3.3					
compiler	0.3	24.3	42.8	16.7	10.1	3.6	1.9	0.1			0.2
conform		20.9	71.7	4.3	3.0						
cpstak		48.5			51.5						
dderiv	1.5	36.6	61.5	0.5							
deriv	2.6	54.8	41.7	0.9							
destruc		4.6	60.1	34.9	0.5						
diviter	2.8	2.8	93.6	0.9							
divrec	1.9	97.5		0.6							
earley		0.1	9.9	0.3	14.3	0.4	0.2		0.1	11.8	62.8
fft		0.2		0.2	11.3	15.0			11.4	61.8	
fftrad4		5.4	25.3					7.5		56.2	5.4
fib		100.0									
fibfp		100.0									
graphs	0.3	13.5	12.4	16.3	13.1			37.1	5.2		2.0
lattice		5.1	68.5	0.2	25.5	0.4	0.1				
mazefun		6.9	76.2	8.2	8.7						
mbrot		5.8				91.2		2.9			
nbody			45.6	54.4							
nboyer		10.5	14.6	64.2	10.6						
nqueens			6.6	85.8	7.7						
nucleic		29.3	57.9	4.2	8.6						
paraffins			32.8	0.3	0.5	63.2	0.4	2.8			
perm9			37.8	32.5	29.6						
peval		41.4	45.4	1.3	0.7		11.1				
pnpoly	0.8			0.1	4.5			94.6			
primes	0.2	5.1	94.7	0.1							
sboyer		9.1	12.7	68.9	9.2						
simplex	0.8	7.7		3.4	8.1	6.5	47.7	2.0	15.0	6.1	2.8
sum			99.9								
sumfp			99.9								
sumloop			100.0								
tak				100.0							
takl			82.5	17.5							
triangl			99.9	0.1							

Tab. 3.1. Percentage of function calls executed per number of arguments for each benchmark.

We obtained these results with our Scheme implementation. It is worth mentioning that our implementation applies optimizations typically used to efficiently implement Scheme that possibly impact the number of arguments used at function calls, in particular *lambda lifting* [**76**]. We can see in the table that in most cases, the functions are called with few arguments. Most of the benchmarks presented in the table are micro-benchmarks that are benchmarking a specific Scheme feature. The *compiler* benchmark is an early version of the Gambit Scheme compiler, it is the biggest benchmark (≈12K lines of code) and the most representative of real world programs. We can see that, for this benchmark, more than 94% of the calls use less than 5 arguments and more than 84% of the calls use less than 4 arguments.

For several benchmarks, most of the functions are called with more than 5 arguments, but this effect is eliminated for all of them if lambda lifting is disabled. Our implementation uses the frontend of the Gambit Scheme compiler thus uses its aggressive lambda lifting approach, suggesting that this optimization is not necessarily suitable for compilers using aggressive lambda lifting.

3.6. Captured Properties

In this section, we explain how we extend our technique to specialize the generated code using captured properties (using $\mathcal{P}_{\text{closure}}$).

In the code we presented in Figure 3.1, the call `(make-sumer 10)` causes the compiler to generate a new version of the `make-sumer` function, specialized for `n` being a fixnum. When generating the specialized closure creation site of `f`, it knows that the free variable `n`, captured by the closures created at this site, is a fixnum. The call `(make-sumer 3.14)` causes the compiler to generate a new version of the `make-sumer` function, specialized for `n` being a flonum. When generating the specialized closure creation site of `f`, it knows that the free variable `n`, captured by the closures created at this site, is a flonum.

However, if the same entry point table is used by all the closure instances of the function `f` regardless of the site from which they are created, the compiler cannot make any assumption on the type of the free variables when generating a version of `f`. It cannot specialize the function `f` using $\mathcal{P}_{\text{closure}}$ and these properties are not propagated and must be discovered again in the body of the function.

To solve this issue, the compiler can use a different entry point table at each specialized closure creation site.

3.6.1. Entry Point Table Specialization

When a specialized closure creation site is generated and the compiler computes a $\mathcal{P}_{\text{closure}}$ that has not been observed before for this function, a new table is created. This means that two closure instances share the same entry point table only if they are created at the same site or from two site that are specialized for contexts that possibly differ, but for which the compiler computed the same $\mathcal{P}_{\text{closure}}$.

Each specialized table has its own set of entry points and its own stub retaining the $\mathcal{P}_{\text{closure}}$ used for the table. When a function is called through a closure and a stub is triggered, it gives its associated $\mathcal{P}_{\text{closure}}$ as an argument to the compiler. The compiler uses it, in addition to $\mathcal{P}_{\text{call}}$, to generate a specialized entry point. The context containing $\mathcal{P}_{\text{call}}$ and $\mathcal{P}_{\text{closure}}$ used for code specialization is then propagated for the rest of the function body as a regular context propagated by BBV.

3.6.2. Example

For the code in Figure 3.1, when the compiler generates the specialized version of `make-sumer` for the call `(make-sumer 10)`, it computes $\mathcal{P}_{\text{closure}} = (n : fixnum)$ for the closure creation site of `f`, creates a table specialized for $(n : fixnum)$, and generates the site creating closures that use this table. When it generates the specialized version of `make-sumer` for the call `(make-sumer 3.14)`, it computes $\mathcal{P}_{\text{closure}} = (n : flonum)$ for the closure creation site of `f`, creates a table specialized for $(n : flonum)$, and generates the site creating closures that use this table.

Figure 3.9 shows the state of these two closure instances after executing the code of the example. The call `(sum-to-10 6)` causes the generation of the first version of `f` specialized for `n` and `x` being fixnums (represented by the lighter gray version). The call `(sum-to-10 7.5)` causes the generation of the second version of `f` specialized for `n` being a fixnum and `x` being a flonum. Finally, the call `(sum-to-pi 1.10)` causes the generation of the third version of `f` specialized for `n` and `x` being flonums (represented by the darker gray version).

Using both interprocedural BBV and table specialization with this example, the type of the argument `x` and the free variable `n` are known in all the versions meaning that all the

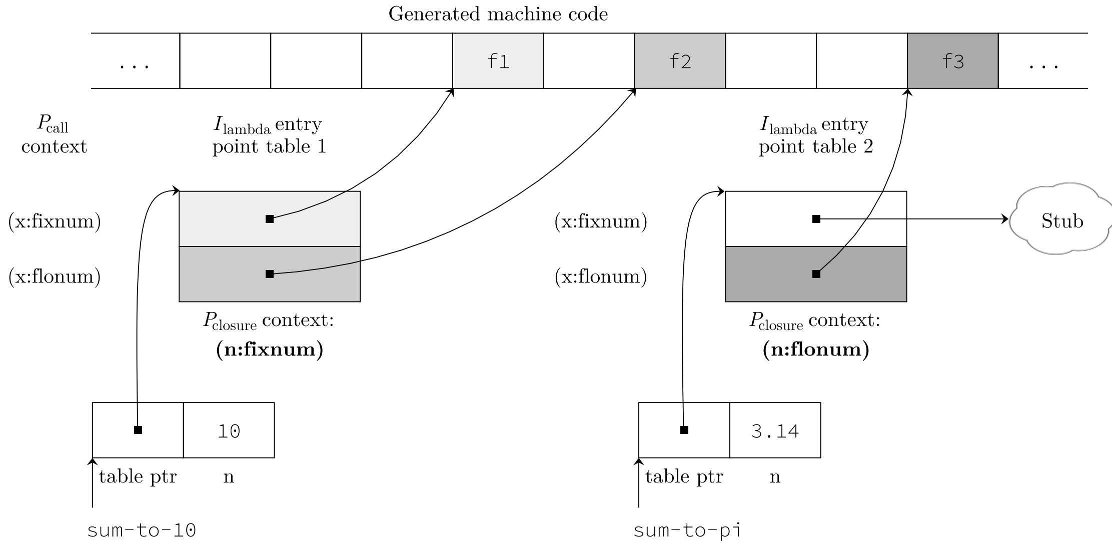


Fig. 3.9. Closures using specialized entry point tables created for the function `f` of the example in Figure 3.1.

tests on these variables are eliminated. The only remaining type test is the one needed to determine the type of the value returned by the recursive call for the `+` operator.

3.6.3. Optimizations

3.6.3.1. *Limiting the Number of Specialized Tables*

Specializing the tables increases memory footprint, especially if many functions with polymorphic free variables are used. We explain in Chapter 6 that the amount of memory used by the tables is not significant, but we still need to prevent pathological cases.

The compiler can set a limit in the number of specialized entry point tables and use a generic table as fallback.

It can also fall back to a table associated with a more general context. For example, if the compiler observes a new $\mathcal{P}_{\text{closure}}$:

$$\mathcal{P}_{\text{closure}} = \textit{(x:fixnum, y:fixnum)}$$

but the limit in the number of specialized tables is reached, then if a table is already specialized for:

$$\mathcal{P}_{\text{closure}} = \textit{(x:unknown, y:fixnum)}$$

it can use it instead of the generic table.

Our experiments suggest that when type information is used to interprocedurally specialize the code of Scheme programs, there is no explosion in the number of table specializations. The left side of table 3.2 shows, for each benchmark, the percentage of tables created per number of specializations. We can see that most of the free variables are monomorphic, meaning that most of the benchmarks require one or two table specializations. These results suggest that setting a limit in the number of table specializations prevents the pathological cases without impacting the optimization opportunities offered by table specialization.

Function entry point tables						Continuation entry point tables					
Number of specializations	**1**	**2**	**3**	**4**	$\geqslant$**5**	Number of specializations	**1**	**2**	**3**	**4**	$\geqslant$**5**
ack	100.0					**ack**	83.3	16.7			
array1	100.0					**array1**	84.2	15.8			
boyer	99.1	0.9				**boyer**	70.2	12.3	1.8	15.8	
browse	91.8	8.2				**browse**	55.9	41.2	2.9		
compiler	95.4	2.8	0.9	0.4	0.5	**compiler**	78.3	14.1	3.8	1.4	2.4
conform	97.8	2.2				**conform**	94.1	5.3			0.6
cpstak	92.2	5.8	1.9			**cpstak**	81.2	18.8			
dderiv	99.0	1.0				**dderiv**	96.4		3.6		
deriv	100.0					**deriv**	95.0		5.0		
destruc	98.0	2.0				**destruc**	84.6	15.4			
diviter	100.0					**diviter**	100.0				
divrec	100.0					**divrec**	100.0				
earley	88.4	3.9	4.5	1.9	1.3	**earley**	78.0	11.0	1.2	3.7	6.1
fft	100.0					**fft**	85.0	15.0			
fftrad4	98.2	1.8				**fftrad4**	98.3	1.7			
fib	100.0					**fib**	100.0				
fibfp	100.0					**fibfp**	100.0				
graphs	92.9	4.5	1.3	0.6	0.6	**graphs**	72.1	11.5	4.9		11.5
lattice	97.0	3.0				**lattice**	69.8	15.1	1.9	7.5	5.7
mazefun	97.8	2.2				**mazefun**	79.3	11.0	2.4	2.4	4.9
mbrot	100.0					**mbrot**	85.7	14.3			
nbody	99.1	0.9				**nbody**	95.5	4.5			
nboyer	89.8				10.2	**nboyer**	65.1	17.4	3.5	12.8	1.2
nqueens	99.0	1.0				**nqueens**	66.7	14.3		9.5	9.5
nucleic	99.4	0.6				**nucleic**	88.9	10.3	0.8		
paraffins	92.9	3.6		0.9	2.7	**paraffins**	62.9	11.4	2.9	2.9	20.0
perm9	99.1	0.9				**perm9**	86.1	13.9			
peval	94.8	3.4	1.7			**peval**	74.9	18.6	3.8	2.2	0.5
pnpoly	100.0					**pnpoly**	89.3	10.7			
primes	98.9	1.1				**primes**	95.5	4.5			
sboyer	89.8				10.2	**sboyer**	65.5	16.7	3.6	13.1	1.2
simplex	100.0					**simplex**	100.0				
sum	100.0					**sum**	81.2	18.8			
sumfp	100.0					**sumfp**	81.2	18.8			
sumloop	100.0					**sumloop**	81.2	18.8			
tak	100.0					**tak**	86.4	13.6			
takl	100.0					**takl**	72.4	17.2	3.4	3.4	3.4
triangl	100.0					**triangl**	90.0	10.0			

Tab. 3.2. Percentage of tables per number of table specializations.

3.6.3.2. *Closure Patching*

When the compiler generates the code of a closure creation site, it is possible that some properties in $\mathcal{P}_{\text{closure}}$ are unknown (e.g. the compiler does not know the type of a free variable). In this case, the table used at this site is not specialized for this property.

However, it is possible that this property is later discovered. For example, the type of an unknown variable can be lazily discovered by a dispatch for a primitive. When this property is discovered, the compiler can patch the current closure using this table to replace it by a table specialized for this property. The table can be created if it does not exist.

It is worth mentioning that the creation site of this closure cannot be patched to use this new table because it is possible that this lazily discovered property differs for the closures created at this site.

3.7. Continuations

CPS represents call continuations using functions. After adding function entry point specialization to our implementation, we explored the implementation of call continuation specialization by transforming the executed program to its CPS representation prior to execution to use our technique to specialize the functions representing continuations.

However, this approach based on code transformation goes against our objectives of limiting the use of IRs and transformations.

The technique used for function entry points can easily be adapted to continuation entry points, allowing specializing the continuations without static transformations.

Continuation specialization is a problem that is analogous to function specialization. For a call continuation:

- $\mathcal{I}_{\text{lambda}}$ is the identity of the continuation
- $\mathcal{P}_{\text{call}}$ is the set of properties known at the function return site used for specialization (e.g. the type of the returned value)
- $\mathcal{P}_{\text{closure}}$ is the set of properties captured when creating the object representing the continuation (e.g. the types of the live local variables)

3.7.1. Continuation Representation Extension

With dynamic languages, continuations are often represented by a single return address (and the call stack). When continuations are specialized, they possibly have several entry points. Their representation must be extended to allow storing several entry points. In a similar way to function entry points, the compiler can use a table to store the continuation entry points. When a table is created, a stub is created for this continuation and the table is initially filled with the stub address.

3.7.2. Dynamic Dispatch

With most programming languages, functions can return a single value only. The benefits of specializing continuations using $\mathcal{P}_{\text{call}}$ is then limited compared to function entry points when applied to typing. Consequently, implementing the dynamic dispatch efficiently is especially important for function returns.

To implement the dynamic dispatch, a global layout can also be used for continuations. The slots in the global layout are lazily allocated each time a new $\mathcal{P}_{\text{call}}$ is observed by the compiler. An extra indirection is added at function returns to retrieve the continuation entry point from the table.

Note that because it relies on the return address, the first solution allowing retrieving the $\mathcal{P}_{\text{call}}$ when a stub is triggered, without generating additional instructions, to generate a specialized function entry point (Section 3.5.2.1) cannot be used for continuations. The two other solutions can however be used for continuations. We explain in Section 3.7.5.1 how the $\mathcal{P}_{\text{call}}$ used at the function return can easily be retrieved when the compiler uses a specific optimization.

3.7.3. Multiple Return Values

Scheme allows functions to return multiple values using the *values* and *call-with-values* functions [**80, 129, 124**]. Ashley and Dybvig [**13**] showed that multiple return values can be efficiently implemented without losing full run-time error checking. That approach creates two entry points in the generated code, next to each other, for each continuation. One entry point is directly used by functions returning a single value. The other is used by functions returning zero or multiple values. When using the second entry point, the number

of arguments is written to a register and is checked by the continuation. Functions returning zero and multiple values are then implemented with low overhead and without penalizing normal calls and returns.

This approach is a form of continuation specialization. One entry point is specialized for functions returning a single value and the other is a generic fallback entry point. Using interprocedural BBV an even more precise specialization is possible without limiting it to functions returning a single value.

If a function f_1 returns n_1 values to a continuation c, the compiler is triggered to generate an entry point of c specialized for the properties propagated for these values. At this point, the compiler then knows n_1. If n_1 is valid for the currently generated continuation, the entry point is generated without needing to insert a run time check to check the number of returned values. If n_1 is invalid, an error is raised.

If a function f_2 later returns n_2 values to the continuation c with $n_1 \neq n_2$, the compiler is triggered to generate another entry point of c specialized for the properties known for these values, including the number of arguments n_2.

Using the same process, each time a new n is used for the continuation c, a new specialized version is generated. No run time check is then needed to check the number of arguments in the code generated for the continuations.

In the approach presented by Ashley and Dybvig, an additional jump is executed each time a multiple-value return point is invoked to jump over the other entry point. Using interprocedural BBV, this jump and the check on number of values returned are eliminated. However, an indirection is added at function returns to dispatch the execution to the specialized continuation entry point.

3.7.4. Captured Properties

For a given continuation creation site, when the compiler generates a specialized version of this site and observes a new $\mathcal{P}_{\text{closure}}$, it can create a new table specialized for this $\mathcal{P}_{\text{closure}}$ and use it at this site. In a similar way to closures, a continuation entry point table is shared by the continuations created at the same site and by the continuations created at sites of the same call possibly specialized for different contexts but for which the compiler computed the same $\mathcal{P}_{\text{closure}}$.

3.7.5. Optimizations

Because the implementation is the same as for function entry points, the optimizations presented in Sections 3.5.3 and 3.6.3 can also be used for continuations.

Regarding the number of times the tables are specialized, we can see in the right side of table 3.2 that, when applied to typing, continuation entry point tables are more specialized than function entry point tables for Scheme programs. Unlike function tables that are specialized for the type of the free variables, continuation tables are specialized for the type of the local variables. Because more local variables than free variables are generally captured, more type combinations are observed at execution. These combinations then require more table specializations. However, this is still not an issue for Scheme programs because they rarely require specializing the tables more than twice. These results also suggest that setting a limit in the number of table specializations prevents the pathological cases without impacting the optimization opportunities for continuations.

3.7.5.1. *Fixed Table Layout*

Depending on the properties used for specialization in the $\mathcal{P}_{\text{closure}}$ set, it is possible to create fixed size continuation tables and to preallocate all the slots before the execution of the program. For example, if the compiler only specializes continuations using the type of the returned value, the exact set of contexts that can be observed at execution is known. It can create tables of a size equal to the number of types it supports with an additional entry for the *unknown* case. Each type is associated to an index in the global layout prior to the execution. Note that using this optimization, when a stub is triggered to generate a specialized continuation entry point, the compiler can easily retrieve the $\mathcal{P}_{\text{call}}$ used at the function return, without generating additional instructions, by retrieving the returned value from the return register and analyzing its type. However, the additional work is still required for the generic case to distinguish it from a specialized case.

If the $\mathcal{P}_{\text{closure}}$ sets contain more properties such as register allocation information or user-defined types, a fixed table layout cannot be used and the compiler must rely on the optimizations presented in Sections 3.5.3 and 3.6.3.

3.7.6. Example

For the code in Figure 3.1, no type test is inserted for `x` and `n` when the compiler specializes function entry points and only one type test remains to determine the type of the value returned by the recursive call.

This value may come from two sites, from the base case of the recursion, or from another recursive call. Using continuation specialization, when the value comes from the base case, its type is known because the type of the constant `0` is propagated through the continuation of the call to `f`. The continuation is then specialized for this type. Because this type is known and the type of `x` is known the type of the result of the addition of these two values is also known and because this value is returned by the function, its type is propagated to the next continuation. Using this mechanism, the type of the returned value is always propagated so that no test is inserted to determine the type of this value when lazily compiling `f`. For this example, all the type tests are then eliminated when function and continuation entry points are specialized.

Note that even when the compiler checks for fixnum overflows, no type test needs to be inserted. The overflow check is itself a control flow dispatch that is lazily generated. Using BBV, the two branches of the result of the check are specialized. Consequently, the compiler always knows the type of the result after the overflow check operation.

3.8. Compiler Optimizations

All the optimizations that can be applied on intraprocedural BBV can be applied interprocedurally, including the optimizations we presented in Section 2.10. However, when the number of propagated properties increases, the compiler may generate code which is overspecialized with an explosion in the code size.

In our implementation, the types are propagated interprocedurally. In Section 4.7, we explain how we perform interprocedural specialization based on register allocation information. We did not observe any significant case of overspecialization in this context.

In our experiments, we tried to use our extensions to interprocedurally propagate constants. We observed a significant positive impact on performance in specific cases. However, in most cases, interprocedural propagating constants resulted in overspecialization. This issue is presented in more details, as well as possible solutions, in Chapter 7.

3.9. Summary

Code specialization is an optimization that is commonly used to efficiently implement dynamic programming languages. Previous research shows that code specialization allows generating more efficient code by decreasing the work done at execution. One of the main limits of code specialization is that it is generally applied intraprocedurally or interprocedurally in a limited form only.

We presented a technique allowing interprocedural code specialization in the presence of higher-order functions working for call sites in which the callee identity is unknown. In addition, the technique allows specializing the code using the properties captured by the closures such as the type of the free variables.

The technique relies on an extension of classical closure representations, an efficient dynamic dispatch based on position invariance and optimizations, in particular *table specialization*.

The technique can easily be extended to specialize call continuations using the same implementation. When applied to typing, using interprocedural code specialization for both functions and continuations allows removing type tests that simpler approaches don't remove, and in some cases all type tests.

Chapter 4

Eager Unboxing Based on Basic Block Versioning

Part of the work presented in this chapter is published in the paper:

- Baptiste Saleil and Marc Feeley. Building JIT Compilers for Dynamic Languages with Low Development Effort. In *Proceedings of the 10th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages, VMIL, 2018.* [**116**]

In this chapter, we expand on the ideas presented in that paper.

4.1. Introduction

BBV opens up the possibility of implementing other optimizations, classically used in optimizing compilers to further decrease the work done at execution. Here we explore how BBV can be used to remove boxing and unboxing operations.

Unlike statically typed languages, dynamically typed languages attach types to values instead of variables. The compiler does not generally know the type of the variables when generating the code. To ensure that the primitives are executed with acceptable types, the compiler must generate code to dispatch the execution using run time type tests when generating primitives.

To implement these type tests, the run time representation of a value includes type information. This grouping of type and value is commonly referred to as a *box*. To determine the type of a value from a box, the compiler must ensure that two values of different types use distinct run time representations.

At an abstract level, three operations on boxes are needed. A *boxing* operation is performed when values are generated. It creates the machine representation of its argument, for example:

$$box_fixnum(n) = machine\ representation\ of\ fixnum\ n$$

```
1:  (define (fib n)
2:    (if (< n 2)
3:        1
4:        (+ (fib (- n 1))
5:           (fib (- n 2)))))
```

Fig. 4.1. Scheme function generating the nth Fibonacci number.

An *unboxing* operation is performed to extract values from boxes when they are needed by primitives:

$$unbox_fixnum(box_fixnum(n)) = n$$

Finally, to dispatch the execution at primitives, *type test* operations are performed to test if a boxed value is of a given type:

$$is_fixnum(box_fixnum(n)) = true$$

In a naive compilation, each time the compiler generates code for a primitive, type tests are inserted to do type checking and dispatch the execution for generic primitives, unboxing operations are inserted to extract the value of the arguments, and a boxing operation is generated to box its result. Chapter 2 explained how BBV can remove the type tests but the issue of boxing and unboxing was not addressed. For instance, for a subtraction of two fixnums (that can be implemented with a single-cycle ALU instruction on most machines), two unboxing operations and one boxing operation are executed in addition to this instruction. The run time cost associated with these operations may vary depending on the box representation the compiler uses, but it can significantly impact the performance of the executed program. In some cases, the box representation the compiler uses completely removes the cost for values of specific types and specific primitives. A common approach is to choose a representation that favors often used types. Compilers can also rely on optimizations to remove boxing and unboxing operations in order to generate more efficient code while ensuring the safety of the language.

Figure 4.1 shows how boxing and unboxing operations can significantly impact the performance of the code generated for a function computing the nth Fibonacci number.

This simple function uses arithmetic operations that compilers generate with few instructions. Using interprocedural BBV, the type tests can be removed. However, 5 unboxing

operations are generated. One to unbox `n` for the `<` operation, two to unbox `n` for the `-` operations, and two to unbox the values returned by the recursive calls for the `+` operation. The compiler also generates 3 boxing operations. Two to box the result of the `-` operations for the recursive calls, and one to box the result of the `+` operation that is returned by the function. Note that we assume the compiler is at least smart enough to not box and unbox the constants. To remove the cost associated with these operations, the compiler can use a representation that favors fixnums such as *Tagging* [**63**] (presented in detail in Section 4.5.1). However, if the `fib` function is called with a flonum, a type that Tagging does not favor, the boxing and unboxing operations will significantly impact the performance of the generated code. Similar issues exist with other representations such as *NaN-boxing* [**123, 50**] (presented in detail in Section 4.5.2) that favors flonums over fixnums. In Section 4.2.1, we present several techniques and the types they favor.

In this chapter, we explain how BBV can be used to easily remove boxing and unboxing operations. We also present the issues related to the implementation of unboxing based on BBV and we explain how they can be solved. We present in detail the implementation, advantages and disadvantages of Tagging and NaN-boxing, the two most used value representations for dynamic languages. We then explain how these representations can be used along with our approach to decrease the cost associated with boxing and unboxing operations for values of all types with relatively low development effort.

4.2. Related Work

4.2.1. Run Time Value Representation

A classic report from 1993 that compares various value representations is Gudeman [**63**]. For several representations, the advantages and disadvantages are presented as well as the cost associated with each operation for computers of the time. Table 4.1 summarizes the cost associated with boxing and unboxing operations for the representations presented in this report for fixnums, flonums and memory allocated objects.

This table also gives an order of magnitude of the cost associated with each operation on current machines. The operations requiring a memory allocation are the most expensive. The operations requiring a memory fetch or write are moderately expensive, but generally more expensive than masking and shift operations on registers. Finally, the operations requiring

	Fixnum		Flonum		Memory allocated object	
Representation	**Boxing**	**Unboxing**	**Boxing**	**Unboxing**	**Boxing**	**Unboxing**
Tagging	masking (and shift)	masking (or shift)	memory allocation	memory fetch	masking (and shift)	masking (or shift)
NaN-boxing	masking	masking	–	–	masking	masking
Partitioned words	–	–	memory allocation	memory fetch	–	–
Object pointers	memory allocation	memory fetch	memory allocation	memory fetch	memory write	–
Large wrappers	use additional location(s)	–	use additional location(s)	–	use additional location(s)	–

Tab. 4.1. Summary of the cost associated with boxing and unboxing operations for various value representations (light colored squares are the least expensive operations).

storing values to additional locations have a different cost depending on whether they use a register or memory location but are significantly less expensive than memory allocation.

Tagging consists of representing a value and its type in a single machine word using a small number of bits to represent the type and the others to represent the value. Depending on whether the tag bits are the most significant or the least significant, fixnums and memory allocated objects are boxed with a single masking operation or with a shift and a masking operation and unboxed with a single masking or shift operation. The main disadvantage of Tagging is that flonums must be memory allocated. This representation favors fixnums over flonums. For fixnums, the boxing and unboxing operations have no cost in many cases if the compiler is careful (see Section 4.5.1).

NaN-boxing consists of using the unused values of the IEEE-754 [**6**] standard used to represent floating point numbers. NaN-boxing favors flonums because they are directly represented by their IEEE-754 representation, no boxing or unboxing operation is needed. The main disadvantages are that it decreases the range of fixnums the compiler can represent and it is less portable than Tagging (details in Section 4.5.2). A similar representation is used in JavaScriptCore [**4**] but it favors memory allocated objects over flonums.

In the *partitioned words* approach, a specific bit pattern is reserved for values of each type. This means that a memory region is reserved for each type to allocate objects of this type. This representation favors fixnums and memory allocated objects because, if the patterns are

wisely chosen, no boxing or unboxing operations on values of these types are needed. The main disadvantages of this approach are that flonums must be memory allocated and, because the objects of each type are allocated in a dedicated region, the allocator performance is negatively impacted.

In the *object pointers* approach, a value is represented by a pointer to a memory block containing both the value and its type. This favors memory allocated objects because they are directly represented by their address, no unboxing operation is needed on these values. The main disadvantage is that all the values need to be memory allocated, significantly degrading performance of fixnums and flonums. However, some local optimizations allow removing the memory allocation needed to box the values in some cases.

In the *large wrappers* approach, more than one machine words are used to represent a boxed value. A common implementation is to use two words, one to store the value and the other to store its type. The advantage of using large wrappers is that, for fixnums and flonums, the value is store unmasked and unshifted in one of the two words meaning that no unboxing operation is needed and boxing is done at low cost. The main disadvantage is that loading and storing a boxed value is slower because more words need to be loaded and stored. Moreover, the representation uses more registers and memory.

We see that each representation favors values of one or more types over the others. Gudeman also presents hybrid approaches combining these techniques to mitigate the negative impact they have on the types they do not favor. In particular, a common approach is to use *Tagging* on least significant bits for non-memory allocated objects and *object pointers* for memory allocated objects to avoid the shift operation when boxing them.

We also see that, depending on the representation the compiler uses, boxing and unboxing operation removal has a different impact on performance. Moreover, it depends on the frequency of use of each type.

Tagging and NaN-boxing tend to be the two most used techniques in recent implementations of dynamic languages. They are presented and quickly compared in several papers presenting new language implementations, in particular for JavaScript [**34, 36, 16**]. However, the representations are not compared in depth in these papers and the implementers typically choose a representation over the others based on intuition without measuring their actual impact on performance.

Caro [**28**] compares NaN-boxing with other techniques such as Tagging using several criteria and comparing the operations on several types.

Unlike the work presented in this chapter, these studies present the representations and explain their impact on performance but they to do not take development effort into consideration and are not supported by experiments.

4.2.2. Boxing and Unboxing Operation Removal

Common Subexpression Elimination [**97, 8**] is an optimization that can be used to remove redundant boxing and unboxing operations. Local CSE can easily be applied because it can be done while generating code for a basic block. Global CSE allows removing boxing and unboxing operations at function level. However, global CSE requires more work to be done statically because they rely on static data flow analysis [**97, 8**]. In contrast, our approach allows removing boxing and unboxing operations globally (and even interprocedurally) without requiring the use of static analysis. In addition, applying CSE in a compiler using the design we present in Chapter 2 is difficult because the compiler uses no IR and has no concept of basic block.

Previous work mentioned that BBV can be used to unbox values [**38**], in particular that interprocedural BBV allows passing function arguments without boxing them. Two main differences with our work is that, as explained in this chapter, we extend this optimization by using a separate register allocation to further optimize flonum arguments and we evaluate the impact on performance with experiments.

Rigo [**108**] explained how specialization by need allows lazily removing boxing and unboxing operations using a lazy code specialization approach that is similar to ours. However, that approach is limited because the code cannot be specialized interprocedurally in the presence of higher-order functions.

Compilers using code specialization with deoptimization typically remove boxing and unboxing operations from the specialized parts of the generated code. Kedlaya et al. [**78**] show the impact of this approach on the number of boxing and unboxing operations in the MCJS implementation. However, these systems rely on complex mechanisms such as OSR to implement deoptimization and the number of removed operations is limited by the scope of the specialization that is often local or global but typically does not apply interprocedurally.

Tracing JITs generate specialized code from traces that contain instructions recorded at execution, following function calls. Tracing thus allows removing boxing and unboxing operations interprocedurally. Gal et al. [**56**] explained that, using Tracing, when the code is generated for a trace, the VM can unbox and copy the values used in the trace to optimize read and write of these variables in the trace and avoid boxing and unboxing operations. The values are then boxed when the trace exits. Bolz et al. [**23**] show how Tracing JIT compilers can further decrease the boxing overhead by removing allocations using partial evaluation. We explained in the previous chapters how Tracing differs from our approach.

4.3. Unboxing Based on Basic Block Versioning

To remove boxing and unboxing operations, the boxing state property (boxed representation or unboxed representation) can be added in the specialization context, in addition to the type. To ensure safety, a variable with unknown type necessarily is boxed meaning that it is represented in the context by the pair of properties *unknown/boxed*. When the compiler discovers the type of a variable, it updates the specialization context with the discovered type. For example, if the variable is a fixnum, the pair of properties is updated to *fixnum/boxed*. While the value is not used, it stays boxed. When the compiler generates the code of a primitive, it first checks the boxing state of the arguments in the context. If an argument is unboxed, it is directly used, no unboxing operation is generated. If it is boxed, the compiler generates code to unbox it and updates the boxing state property in the context to unboxed. Once the unboxing code is generated for a fixnum, the pair of properties is updated to *fixnum/unboxed*. With BBV, the subsequent blocks are specialized for these two properties. When another primitive in a subsequent block uses this value, the compiler knows its type and knows that it is unboxed, and no type test or unboxing operation needs to be generated. The value is maintained unboxed as long as the compiler knows its type. When the type is lost, for example when the limit in the number of specialized versions of a block is reached, a boxing operation must be generated. When the value is copied to a compound data type for which the compiler does not track the types, it generates a boxing operation to box the copy of the value. Note that if the value is copied several times in a compound data type, it will be boxed several times. A solution to this issue would be to propagate the value

both in its boxed and unboxed representations. This approach avoids generating additional boxing operations but increases register pressure.

The compiler often knows the type of the values generated by the language primitives. For example in Scheme, when the compiler generates code to inline the *fl+* primitive (flonum addition), it knows that the result is a flonum and it is unboxed because there is no immediate need to box it. The pair of properties *flonum/unboxed* is then added to the context and used to specialize the subsequent blocks without boxing or unboxing.

Using this mechanism, when the compiler knows that the arguments of a primitive are unboxed, and knows the type of its result, no boxing or unboxing operations are generated.

4.3.1. Example

Here we explain how unboxing based on BBV is efficient at removing boxing and unboxing operations for the `fib` function presented in Figure 4.1. We assume that when the function is called, a fixnum or flonum argument is passed but the compiler does not yet know the type of the argument. A type dispatch is generated before generating the `<` operation. Once the type of `n` is discovered from the dispatch (either fixnum or flonum), it is added to the context. The property *boxed* is also added to the context.

When generating the `<` operation, the compiler determines from the context that its argument `n` is boxed, it generates an unboxing operation, generates the code for the primitive and updates the boxing state of `n` to *unboxed* in the context (either *fixnum/unboxed* or *flonum/unboxed*).

Because the following code is specialized for both the type and the boxing state, the variable `n` is directly handled unboxed, no more unboxing operation on `n` is inserted for the `-` operations.

Using interprocedural BBV, the types are propagated through function calls. Assuming that the compiler does not handle fixnum overflows (we will lift this constraint later), it knows the type of the result of the `-` operations. The result of the `-` operations are then propagated unboxed through the recursive calls. When a recursive call is executed, it then triggers the generation of a new version of `fib`, specialized for the properties of `n` (i.e. *fixnum/unboxed* or *flonum/unboxed*).

When the base case is reached, `1` is returned unboxed because, using interprocedural BBV, the compiler is able to propagate its type and its boxing state to the call continuation. In the same way, the type of the result of the `+` primitive is known by the compiler and is passed unboxed to the call continuation.

When generating the `+` primitive, the compiler knows the type of the two arguments, and it knows that they are unboxed. No unboxing operation is generated.

Consequently, for the `fib` function of this example, a single unboxing operation is needed in the generic version of the function. When this version is executed, the type of the argument is discovered at the first call and all the subsequent recursive calls use a version specialized for this type with no boxing or unboxing operations, regardless of the value of `n`.

Note that a single type test and unboxing operation is executed even if the type of `n` or the result changes during the computation. This can happen when the compiler handles fixnum overflows, the function is called with a fixnum, the `+` operation overflows and the compiler generates code to use a different representation for the result (bignum or flonum). In this situation, the compiler knows the new type and propagates it, causing the generation of a new version of `fib` specialized for this type with no boxing or unboxing operation.

4.4. Eager Unboxing

To decrease the development effort, and simplify the context representation, the compiler can use a heuristic that eagerly unboxes the values. Instead of propagating the boxing state of a value, and unbox it when it is first used, the compiler can unbox it as soon as it discovers its type, speculating that the unboxed value will soon be needed. In other words, a variable with unknown type is always boxed and a variable with known type is always unboxed.

In the example of Figure 4.1, this heuristic achieves the same result, a single unboxing operation is generated. However, the unboxing operation is generated as a prologue to the branch generated when the type dispatch discovers the type of `n` instead of at the `<` primitive.

Figure 4.2 shows an example where the heuristic generates an unnecessary unboxing operation. When the function `f` is called with a value of unknown type, the value is initially boxed. When the compiler discovers its type with the primitive `pair?`, the heuristic causes the generation of an unboxing operation. However, this value is not used after its type is discovered. This unboxing operation is then unnecessary and will be executed each time `f` is

```
1:  (define (f x)
2:    (if (pair? x)
3:        1
4:        0))
```

Fig. 4.2. Example of code causing eager unboxing to generate an unnecessary operation.

called with a value of unknown type. Note that in this specific example, a simple liveness analysis would help the heuristic avoid the generation of this additional operation.

4.5. Comparison of Tagging and NaN-boxing

Compilers that do unboxing based on BBV (whether they use the eager unboxing heuristic or not) can choose to restrict the types on which it is performed. In particular, if the representation used by a compiler makes the cost of boxing and unboxing values of specific types very low there is no incentive to remove the operations.

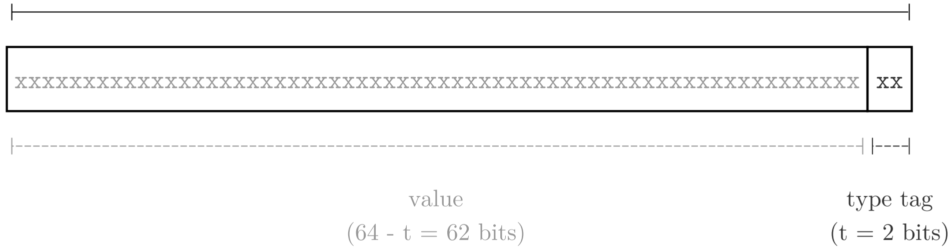
Consequently, it is a good design strategy to choose a representation that favors a frequently used set of types and focus development effort on removing boxing and unboxing operations of the other types with BBV.

In this section, we present in detail Tagging and NaN-boxing, that are the two most used representations to implement dynamic languages. We explain which one would be the best choice to use in a simple compiler based on BBV which is designed to achieve good performance with low development effort. Our explanations are in the context of Scheme but this work is applicable to other dynamic languages with only minor adjustments.

4.5.1. Tagging

Tagging consists of using t bits of a machine word of size n representing a value to encode the type. The remaining n-t bits are used to store the value. Although the most significant bits can be used to encode the type, the least significant bits are generally used because it simplifies some operations (presented later in this section). Few bits are typically used for the type to use as many bits as possible to encode the value.

Machine word (n = 64 bits)



Examples

fixnum 42	`0000000000000000000000000000000000000000000000000000000010101000`
boolean false	`0000000000000000000000000000000000000000000000000000000000000010`
boolean true	`0000000000000000000000000000000000000000000000000000000000000110`
Memory object allocated at address 140737354096688	`0000000000000000011111111111111111110111111111110110000000110001`

Fig. 4.3. Tagging representation and example of boxed values with *n=64* and *t=2*.

In the rest of this chapter, we use the term Tagging to refer to the hybrid approach presented in Gudeman's report as *Tagged Object Pointers* that is typically used to implement Tagging. We also consider that the type tag is written in the least significant bits.

The top side of Figure 4.3 shows a typical implementation of Tagging using 2 bits to represent the type on 64 bits words (*n=64, t=2*). *t=2* is a popular choice and is used by our compiler. The bottom side of this figure shows the encoding of a few Scheme values using this implementation of Tagging by using the tag `00` for fixnums, `10` for booleans and `01` for memory allocated objects.

In this section, we present, for each type, the strategies that can be applied to generate efficient code for the boxing and unboxing operations. We also present the impact of Tagging on the values of each type.

The type checking operation is implemented in the same way for values of all types. The t bits of the machine word must be extracted to retrieve the type associated with a value. For Tagging, type checking is implemented using a binary mask of length t to extract the type and comparing it with the expected type. For memory allocated objects, a secondary

$$
\begin{array}{rrl}
 & 101010\,|\,00 & \text{(encoded } 42_{10}\text{)} \\
+ & 1100100\,|\,00 & \text{(encoded } 100_{10}\text{)} \\
\hline
= & 10001110\,|\,00 & \text{(encoded } 142_{10}\text{)}
\end{array}
$$

Fig. 4.4. Example of a fixnum addition with no unboxing operations.

Boxing	**Unboxing**
`sal rax, t`	`sar rax, t`

Fig. 4.5. Implementation of boxing and unboxing operations for tagged fixnums (using the tag `0`) on x86_64 machines (Intel syntax). The fixnum is initially in the `rax` register.

type information can be stored in the header of the object, thus requiring one tag test, a memory read of the header, a mask and comparison.

4.5.1.1. *Fixnums*

Using Tagging decreases the range of fixnums that can be represented. For example, when *t=2*, 62 bits are available to encode the value instead of 64.

A compiler using Tagging generally uses the tag `0` to encode fixnums. Using this tag allows performing boxing and unboxing operations on fixnums at no cost for most of the operations. Figure 4.4 shows an example of an addition of two fixnums encoded with this tag. By performing the addition on the boxed representations, the result is itself already boxed; no boxing or unboxing operation is needed for this operation. However, for some operations, at least one of the primitive's arguments must be unboxed. For example to multiply two fixnums encoded with the tag `0`, one of the operands must be unboxed before performing the multiplication. However, the result of this multiplication is correctly tagged, so there is no boxing cost. In this case, a single unboxing operation is required instead of three operations.

Unboxing a fixnum is a single right shift by t of the machine word. On x86_64 machines, this is performed with a single instruction. Figure 4.5 gives the implementation of boxing and unboxing operations for fixnums on x86_64 machines.

4.5.1.2. *Special Values*

A special value is a value of a type having a fixed small set of instances such as the Scheme booleans *true* and *false* and the *null* value. The run time representation of special values is not an important concern because they don't need to be unboxed.

These values are generally used as constants or for run time checks. In case they are used as constants, the compiler can directly use the boxed constant representation. In case they are used for checks, the checked value can directly be compared to the boxed constant representation.

4.5.1.3. *Flonums*

Flonums are the values for which Tagging impacts performance most. The IEEE-754 standard for floating point arithmetic requires the use of 64 bits to represent a double precision floating point number. However, with Tagging, because only $64 - t$ bits are available to represent the value, a flonum implementing the IEEE-754 standard cannot be represented using a single machine word.

To solve this issue, compilers using Tagging allocate flonums in memory and represent the number with a tagged pointer, as a regular memory allocated object. Because flonums are memory allocated, the boxing operation adds considerable overhead because a memory allocation and a memory write are executed for each operation. Unboxing operations also are expensive compared to other types because a memory fetch is executed for each operation.

Figure 4.6 gives the implementation of boxing and unboxing operations for flonums on x86_64 machines using a "best-case" bump allocator with the allocation pointer pre-assigned to a machine register and a specific tag for flonums.

4.5.1.4. *Memory Allocated Objects*

For memory allocated objects, the tag bits are typically directly written in the t least significant bits of the object address without shifting it. To ensure that these bits are available for the tag, the memory allocator word aligns addresses when allocating objects. A memory allocated object is then represented by a machine word containing the value:

$$address + tag_mem$$

Boxing	**Unboxing**

`add r8, 8`
`cmp r8, r9`
`jle success`
`call gc`
`success:`
`movsd [r8 - 8], xmm0`
`lea rax, [r8 - 8 + tag_flo]`
	`movsd xmm0, [rax - tag_flo]`

Fig. 4.6. Implementation of boxing and unboxing operations for tagged flonums on x86_64 machines (Intel syntax). The flonum is initially in the `xmm0` floating-point register for the boxing operation and in the `rax` register for the unboxing operation. The boxing code uses a bump pointer allocator. `r8` is the heap pointer and `r9` is the heap limit address.

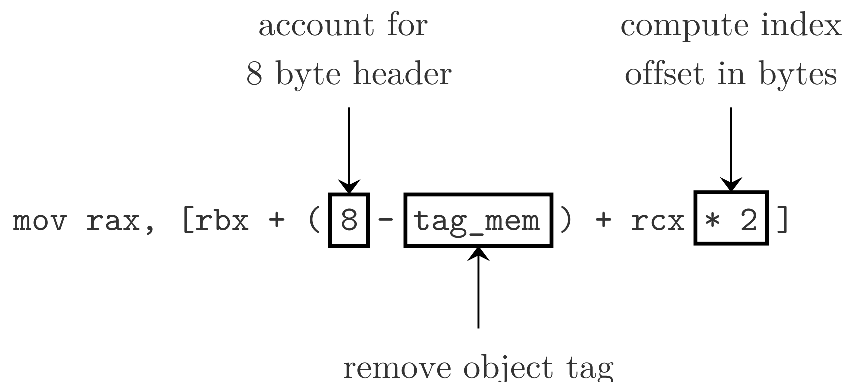


Fig. 4.7. Implementation of the Scheme `vector-ref` primitive using a single x86_64 instruction. `rax` is the destination register, `rbx` is the boxed vector and `rcx` is the boxed fixnum index.

with *address* the aligned address of the object in memory and *tag_mem* the value of the tag representing memory allocated objects. The compiler can exploit this representation to perform boxing and unboxing operations at low cost on specific architectures, including x86_64. On these machines, the addressing mode allows using a constant (*displacement*) when addressing memory. When accessing an object, this displacement can be used to effectively remove *tag_mem* from the word and perform the object access using a single instruction.

Figure 4.7 shows how Tagging can use the addressing mode of x86_64 to perform the Scheme call `(vector-ref v i)` to get the `i`th element of the vector `v` using a single instruction.

We assume that the compiler does not check vector boundaries and that two bits are used for the tag. For this example, `rbx` is the tagged vector and `rcx` is the tagged fixnum index. The addressing mode allows accessing the $\texttt{i}^{th}$ element without requiring any unboxing operation. `8` is added to the word to skip the vector header. `tag_mem` is subtracted to remove the type tag. `rcx` is multiplied by `2` to compute the offset of the element from its tagged index given that each element is 8 bytes. Because two bits are used for the tag, and the fixnum tag is `00`, the multiplication by 2 of this fixnum gives the offset in bytes corresponding to the index. After executing the instruction, the register `rax` contains the tagged $\texttt{i}^{th}$ element. No explicit boxing or unboxing operation is generated.

4.5.1.5. *Summary*

To summarize the use of Tagging:

- **fixnum:** no boxing/unboxing cost in most cases
- **special values:** no boxing/unboxing cost in most cases
- **flonums:** high boxing/unboxing cost, due to memory allocation
- **memory allocated objects:** no boxing/unboxing cost possible in popular architectures

Tagging is a simple technique, both conceptually and to implement. The technique is portable because it depends on basic features of the target machine (masking, shifting, adding constants). The addressing mode of the machine can be exploited, if it allows the use of a displacement, to generate more efficient code but this is not required by the representation. We see that Tagging allows compilers to optimize boxing and unboxing operations on values of all types except flonums. Consequently, BBV based unboxing can be restricted to flonums.

4.5.2. NaN-boxing

NaN-boxing is based on the property that in the IEEE-754 standard, a big range of values is reserved for representing NaNs (Not a Number, the representation of undefined values). A compiler using NaN-boxing directly represents flonums by their IEEE-754 representation and encodes the values of other types in the unused portion of the NaN space.

In the double precision IEEE-754 standard, a flonum is represented by a 64-bit machine word with a bit of sign, 11 bits of exponent and 52 bits of fraction (from the most significant to the least significant). A word represents a NaN if all the exponent bits are set and the

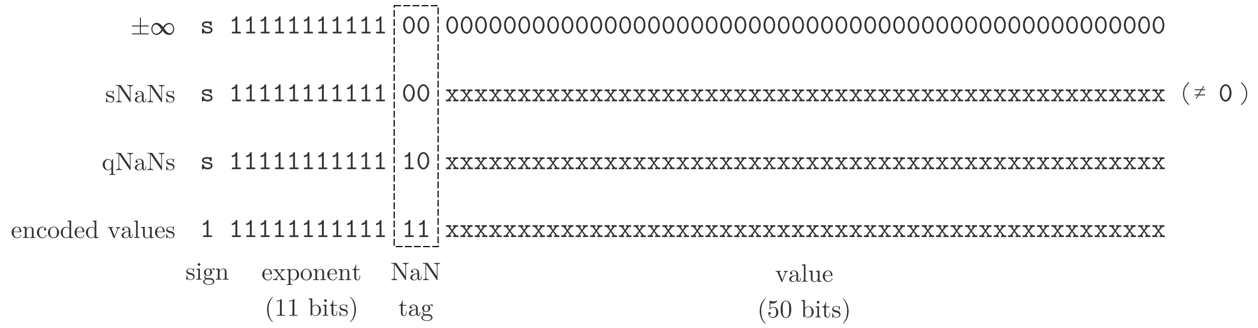


Fig. 4.8. Example of canonical representations for NaNs.

fraction is not 0. In the IEEE-754 standard, there are two types of NaNs, namely signaling and quiet NaNs (sNaN and qNaN).

Compilers can collapse NaN values into canonical representations to increase the number of unused NaNs thus to increase the number of values available to encode values of other types. If a compiler needs to distinguish sNaN from qNaN, it must then use at least two canonical NaN representations.

Figure 4.8 shows a possible implementation using a two-bit NaN tag to represent sNaNs and qNaNs. For a given machine word, if the exponent bits are all set and the NaN tag is 10, it represents a qNaN. The tag 00 allows representing the IEEE-754 infinities (if the value is 0) and the sNaNs (if the value is not 0). The tag 11 is used to encode the values of other types.

Compilers can adapt the implementation to the target machine for performance. For example, the representation in Figure 4.8 is particularly well adapted to the x86_64 architecture [**72, 7**] because it requires almost no conversion of NaNs to their canonical representation at run time. The qNaNs generated by these CPUs have the two bits of the NaN tag of this representation set to 10 meaning that no check nor conversion is needed.

These CPUs consider a NaN as sNaN if the most significant bit of the fraction is clear. However, they do not generate sNaNs, they are generated by software only. Consequently, the compiler has control over the sNaNs used at execution time meaning that it can safely convert them to their canonical representation before using them to ensure they use the tag 00.

In some cases, these CPUs may convert a sNaN to a qNaN. For this conversion, the CPU sets the most significant bit of the fraction to 1. The tag 01 is not used to represent the sNaNs to avoid collisions between encoded values and CPU converted sNaNs. Using this

Machine word (64 bits)

x	xxxxxxxxxxx	xx	xx	xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
sign	exponent (11 bits)	NaN tag	type tag	value (48 bits)

Examples

fixnum 42	1111111111111100 000000000000000000000000000000000000000000101010
boolean false	1111111111111110 000000000000000000000000000000000000000000000000
boolean true	1111111111111110 000000000000000000000000000000000000000000000001
Memory object allocated at address 140737354096688	1111111111111101 001111111111111111110111111111101100000000110000
flonum 3.14	0100000000001001 000111101011100001010001111010111000010100011111
qNaN 0	x111111111111000 000000000000000000000000000000000000000000000000

Fig. 4.9. NaN-boxing representation and example of boxed values.

implementation, NaN value checks and conversions are inserted in the generated code only when an external library is used (e.g. using a *Foreign Function Interface* (FFI) function call).

The compiler can then safely use the tag `11` to encode the values of other types. This means that, in this implementation of NaN-boxing, if the 14 most significant bits are set in a word, it represents an encoded value. Using this kind of representation, the compiler is able to distinguish encoded values from regular flonums, including NaNs and infinities. It can then use the remaining bits to encode other values and their types.

The top side of Figure 4.9 shows the details of a full implementation of NaN-boxing for a dynamic language. We see that two bits are used for the type tag in addition to the NaN tag. The remaining 48 bits are used to store the value. The bottom side of this figure shows Scheme values encoded by this implementation of NaN-boxing by using the tag `00` for fixnums, `10` for booleans and `01` for memory allocated objects. The most significant bits are generally used to encode the type because it allows storing the raw value in the remaining

bits, without needing to shift it. NaN-boxing also exploits the fact that on specific systems, the addressing space is limited to 48 bits so the address of a memory allocated object fits in 48 bits and can safely be written after the type tag.

There are several ways to implement NaN-boxing by using the bits in various ways to represent the NaNs and the value types. However, all the implementations are based on a similar approach.

With NaN-boxing, the type checking operation is implemented in a similar way for values of all types. Using the implementation outlined in Figure 4.9, type checking can be implemented for all types except flonums by extracting the first 16 bits and comparing them with the bits of the expected type. To perform a flonum type check, the compiler first extracts the 14 most significant bits (the sign, the exponent and the NaN tag) and checks if all of these bits are set. If it is not the case, the word represents a flonum. The cost of type checking with NaN-boxing is then similar to the cost of type checking with Tagging (mask plus comparison to a constant).

Because the number of bits available to store the values may vary depending on the implementation of NaN-boxing, we refer to this number as b_{value} in the rest of this section.

4.5.2.1. *Fixnums*

Using NaN-boxing, a fixnum can only be represented using b_{value} bits. With current CPUs, it is typically not possible to perform fixnum operations with overflow checking on fixnums represented by an arbitrary number of bits. For example, if the compiler represents fixnums on 48 bits on x86_64 machines, the compiler must use 64 bits operands to perform addition of fixnums and the *Overflow Flag* [**72, 7**] cannot be used to detect overflows. The overflow check must then be performed before the operation, which adds overhead.

Another solution available on specific architectures is to use 32-bit fixnums. The range of fixnums the compiler can represent is significantly decreased.

However, an advantage of representing fixnums on 32 bits is that some instruction sets allow accessing the low 32 bits of 64-bit registers. It allows unboxing at no cost in most cases because the low 32 bits of NaN-boxed fixnums can directly be used as operands of arithmetic instructions. The result still needs to be boxed.

Boxing	**Unboxing**
`mov rbx, mask_fix` `or rax, rbx`	`mov eax, eax`

Fig. 4.10. Implementation of boxing and unboxing operations for NaN-boxed fixnums on x86_64 machines (Intel syntax). The fixnum is initially in the `eax` register (lower 32 bits of the `rax` register). The `mov` instruction in the unboxing code zero-extends the value in `eax` to `rax` (the 32 most significant bits are cleared and the 32 least significant bits, representing the fixnum, are unchanged). However, the unboxing operation is typically not needed on x86_64 because the 32-bit instructions can directly use the 32-bit registers (including `eax`). Note that the `cdqe` instruction can be used if the value needs to be sign-extended instead of zero-extended.

A boxing operation can be implemented by modifying the $64 - b_{value}$ most significant bits of the result to write the bits representing the fixnum type. Figure 4.10 gives the implementation of boxing and unboxing operations for fixnums on x86_64 machines.

4.5.2.2. *Special Values*

The cost of boxing and unboxing special values is not relevant because these operations are rarely executed. With NaN-boxing, special values can be written in the b_{value} bits and use their own tag. The compiler can then directly use the boxed value if it is used as a constant and it can perform a comparison with the boxed constant if the value is used for a run time check.

4.5.2.3. *Flonums*

The main advantage of NaN-boxing is that flonums are directly represented by their IEEE-754 representation. Unboxing operations are performed at no cost because the arguments of the flonum primitives already are represented by their IEEE-754 representation. The boxing operations are also performed at no cost because the result of the flonum primitives generated by the processors implementing the IEEE-754 standard are already represented by this standard.

Boxing	**Unboxing**
`mov rbx, mask_mem` `or  rax, rbx`	`mov rbx, 2^48 - 1` `and rax, rbx`

Fig. 4.11. Implementation of boxing and unboxing operations for NaN-boxed memory allocated objects on x86_64 machines (Intel syntax). The object is initially in the `rax` register. We assume that the address of the object fits in 48 bits and is positive. For the unboxing operation, the constant is loaded with a `mov` instruction because the other instructions limit the constants to 32 bits.

4.5.2.4. *Memory Allocated Objects*

Because NaN-boxing relies on the fact that the current systems address memory on 48 bits, the address of memory allocated objects can be written in the 48 lower bits of machine words and the $64 - b_{value}$ bits are modified to represent the memory allocated object type. When an object is accessed by a primitive, an unboxing operation is required. Unboxing a memory allocated object then consists of clearing the 16 most significant bits. Unlike Tagging, the constant in these 16 bits cannot be removed by exploiting the addressing mode on x86_64 machines because the displacement constant is limited to 32 bits. This means that additional instructions must be generated to unbox the object before accessing it.

For a boxing operation, the compiler must generate the code to write, in the 16 most significant bits, the bits representing the memory allocated object type. For the same reason as the unboxing operation, additional instructions must be generated to box an object.

Figure 4.11 gives the implementation of boxing and unboxing operations for memory allocated objects on x86_64 machines.

4.5.2.5. *Summary*

To summarize the use of NaN-boxing:

- **fixnums:** no unboxing cost for 32-bit fixnums when using the 32-bit instructions, but the primitive results must be boxed.
- **special values:** no boxing/unboxing cost
- **flonums:** no boxing/unboxing cost

- **memory allocated objects:** boxing/unboxing operations require additional instructions

NaN-boxing is a technique that is more complex than Tagging and relies on clever tricks. This representation is not portable because it relies on features of specific processors and systems for NaN representations and addressing space. As an example, the virtual address space of Solaris on x86_64 includes addresses represented by more than 48 bits [**91**] which makes the use of NaN-boxing more difficult. Compared to Tagging, no type is significantly more expensive to box and unbox than the others with NaN-boxing. However, the overall boxing and unboxing cost is distributed among the types. In particular, unlike Tagging, additional instructions are required to box fixnums and to box and unbox memory allocated objects which are intensively used by Scheme programs.

4.5.3. Choosing a Representation to Decrease Development Effort

The goal of this exploration is to choose the representation that would allow BBV achieving the best performance to get as close as possible to the performance of highly-optimizing state-of-the-art JIT implementations, without significantly increasing development effort.

Consequently, for this context, we think that Tagging is better than NaN-boxing. Unlike NaN-boxing, the cost of the boxing and unboxing operations is not distributed among all the types. The values of all types except flonums already are optimized with Tagging, but the performance of flonums is much more impacted than with NaN-boxing. The compiler writer can then focus the development effort on flonums only and implement unboxing based on BBV on the values of this type only.

Tagging also is simpler and more portable than NaN-boxing. Because it is more portable, less development effort is required in case the support of a new target machine with different characteristics is added.

In addition, in the single precision IEEE-754 standard, 23 bits are used for the fraction, which significantly decreases the number of values that can be represented (including the range of fixnums and virtual addresses) meaning that NaN-boxing is basically inapplicable to 32-bit boxes.

To summarize the use of Tagging with flonum unboxing based on BBV:

- **fixnums:** no boxing/unboxing cost in most cases

- **special values:** no boxing/unboxing cost in most cases
- **flonums:** a significant number of boxing/unboxing operations are removed by unboxing based on BBV
- **memory allocated objects:** low boxing/unboxing cost if the addressing mode allows it

In Chapter 6 we present a performance comparison of pure Tagging, NaN-boxing and Tagging used with flonum unboxing based on BBV. Tagging with BBV flonum unboxing has better performance than NaN-boxing for almost all of the benchmarks.

4.6. Garbage Collector

Using unboxing based on BBV, the values with known types are lazily unboxed meaning that the registers and the execution stack may contain both boxed and unboxed values. The GC then cannot use a blind sequential stack scanning approach because only boxed values count as roots.

As explained in the previous section, only flonums are unboxed in our implementation of unboxing based on BBV. We must then add the ability to the GC to distinguish boxed flonums (that are GC roots) from unboxed flonums (that are not GC roots) when scanning the stack.

In the rest of this section, we explain how the issue can be solved for flonums, but the same solutions can be applied in case unboxing based on BBV is used on values of other types.

4.6.1. Separate Stack

A simple solution to solve this issue is to use a separate stack to store the unboxed flonums. When the compiler decides to spill an unboxed flonum, it allocates it to this separate stack instead of the main program execution stack. When the GC is triggered, it scans the main execution stack only because it knows that all the slots in the separate stack contain unboxed values that are not roots.

However, the solution impacts the performance of the generated machine code because the compiler must keep both stacks updated. For instance, to pop a stack frame at a function return, two stack pointers must be updated if the frame contains boxed and unboxed values.

4.6.2. Garbage Collection Stack Maps

Another solution is to use GC stack maps [**77**]. With GC stack maps, a frame descriptor is inserted in the generated code at each return point. This descriptor indicates which stack slots the GC must scan in addition to the frame size. In our implementation, each stack slot is represented in the descriptor by a single bit which is set if the compiler knows that this slot contains a value that is unboxed. If the bit is not set, the slot necessarily contains a boxed value the GC can safely scan. Each time the GC is called, an initial descriptor containing information on the first frame is given to it. Once all the slots indicated by the descriptor are scanned, the GC retrieves the return address of the current frame from the frame size field of the descriptor. It then retrieves the descriptor of the next frame from this return address. The same process is repeatedly applied until all the frames are scanned.

If the interprocedural code specialization extensions are used, call continuations are represented by continuation entry point tables instead of return addresses. Consequently, the compiler can write descriptors in the continuation entry point tables. A new frame descriptor is then created and stored each time a new specialized continuation entry point table is created.

When the GC is called, the first frame is scanned using the initial descriptor. The descriptor of the next frame is retrieved from the continuation entry point table of the current frame. The same process is then applied until all the frames are scanned.

Using GC maps does not impact the performance of the generated machine code.

In the implementation presented in Chapter 2, the current frame is represented by a stack of types in the specialization context. The frame descriptor is then easily computed from the context.

4.6.3. Conservative Garbage Collection

Another solution to solve this issue is to use Conservative Garbage Collection [**21, 77**]. Using a conservative GC, the stack can be scanned without knowing the locations of the unboxed values. When the stack is scanned, all the slots are considered as possible memory allocated objects. The GC then uses additional checks to determine if a scanned slot actually is a memory allocated object. These checks include checking if it contains an address that is

inside the heap, and checking that the address actually points to a valid memory allocated object.

Using a conservative GC, it is possible that some non-reachable memory allocated objects are retained by the GC. This is for example the case if a slot in the stack contains an unboxed value that is not a memory allocated object (e.g. a flonum in our implementation) and it turns out that this value represents a valid unreachable memory allocated object address. However, conservatively retaining more objects than the reachable ones is not a problem in practice as long as all the reachable objects are retained.

Using a copying GC, if a stack slot that is not a memory allocated object is conservatively considered as one, the GC will move the object during the moving phase and the value in the stack slot will be corrupted. The conservative approach is thus incompatible with Gambit's copying GC. Because our implementation uses Gambit's GC, we instead decided to use the GC stack maps approach.

4.7. Register Allocation Extension

On specific architectures, specialized registers and instructions are used for flonum computations. This is the case with x86_64 machines supporting SIMD extensions (SSE) with the *xmm* registers. Using naive compilation, cross-domain moves (moves between general and floating-point registers) need to be executed in addition to unboxing code for each flonum operation. For example, for a flonum addition, the two arguments must first be unboxed and moved to floating-point registers. The addition is then performed, the result is moved back to a general register and it is boxed.

When the compiler uses unboxing of flonums based on BBV, it can use a separate register allocation using floating-point registers to handle unboxed flonums. When the compiler discovers that a value is a flonum and unboxes it, it allocates it to a floating-point register. This value is then handled unboxed by the separate register allocation while its type is known. If its type is lost, the flonum is allocated back to a general register. It is then boxed and handled by the general register allocation while its type is unknown. Using this separate register allocation then allows generating more optimized code because, when control flows to a flonum primitive, its arguments already are unboxed and in the floating-point registers.

In the same way, the flonum primitives results already are unboxed and in floating-point registers. The compiler can then propagate these properties to specialize the following code.

For flonum intensive Scheme programs, because interprocedural BBV allows our implementation to know the types in most cases, most of the operations on flonums directly use the floating-point registers, without requiring unboxing or cross-domain moves, significantly improving the overall performance.

In addition, if the types are propagated interprocedurally, the unboxed flonum arguments and the unboxed flonums returned can be given in the floating-point registers to the callee functions and continuations, meaning that no boxing nor cross-domain move is even required at function calls and returns. This is a form of interprocedural specialization based on a "register allocation" property.

When Tagging is used, when a flonum is unboxed, it is first extracted from the memory box, then moved to a floating-point register. When NaN-boxing is used, because flonums already are represented by their IEEE-754 representation, when a flonum is unboxed, it is moved to a floating-point register with a cross-domain move.

4.8. Summary

BBV allows discovering types and allows unboxing the values to decrease the cost of the boxing and unboxing operations, with low development effort. Using a simple heuristic, values can be eagerly unboxed. With our approach, the values with known types are handled unboxed while their type is known.

Our analysis of Tagging and NaN-boxing, the two most used run time value representations, showed that Tagging, used with flonum unboxing based on BBV, offers the best trade-off between the types that are naturally optimized by the representation and the effort required to optimize the others.

Finally, BBV based flonum unboxing can benefit from extending the register allocator and specializing the function and continuation entry points with register allocation properties to use the floating-point registers available on specific architectures and avoid cross-domain moves.

Chapter 5

LC

In this chapter, we explain the development of a JIT compiler for a programming language offering a set of features similar to the Scheme programming language, based on the techniques presented in the previous chapters.

We evaluate the effort required to build LC, our Scheme implementation, and we present the tricks and techniques we used to minimize the development effort.

This chapter presents our design and explains our trade-off between simplicity/development effort and performance/set of features the implementation provides.

5.1. Overview

The techniques presented in the previous chapters have been implemented in LC, a JIT compiler for the Scheme programming language. LC is itself written in Scheme (using the Gambit Scheme dialect). LC implements a subset of the R5RS Scheme standard [**80**]. Missing R5RS features are not implemented because they are not necessarily relevant for our research but the architecture of the compiler and the techniques it uses do not prevent their implementation. The two main missing features are first-class continuations (i.e. the `call/cc` operator) and full support of `eval` (only a limited form is currently supported). Although we explained in Chapter 3 that BBV can be used to efficiently implement multiple return values, it is not implemented in LC. The JIT compiler emits x86_64 machine code on-the-fly and targets GNU/Linux. It is likely that LC works as is on other systems in great part because Gambit is highly portable.

LC is available at [**110**] and licensed under the 3-Clause BSD License [**98**].

5.2. Implementation From Scratch

LC first aims to validate the use of BBV for dynamic programming languages offering a set of features similar to the Scheme programming language. These features include the extensive use of lists, higher-order functions and proper handling of tail calls. The second objective of LC is to validate the techniques and optimizations presented in this thesis. It shows that these techniques can actually be used to implement a JIT for a language offering this set of features and allows us to analyze their impact on performance of the generated machine code and on the development effort.

We decided to start a new compiler project instead of extending an existing implementation. Starting from an existing compiler would allow better support of the features of the source language. However, because the implemented techniques are at the core of the compiler (in particular BBV, the compilation design, Tagging, NaN-boxing and interprocedural code specialization), too many changes would have been required.

Building a new compiler allows us to have more control and more freedom of implementation choices. In addition, building an implementation from scratch, that is based on the techniques presented in this thesis, helps to evaluate their impact on development effort.

LC first used its own GC. However, to avoid the use of multiple GCs, we later decided to use Gambit's GC, adding a dependency to its runtime system. More details are given in Section 5.5.

We started working on LC in the summer of 2014, it contains around 20K lines of Scheme code, which is low for an optimizing JIT compiler and is built mainly by a single person.

Our techniques and optimizations allow LC to generate optimized code that is often more efficient than existing AOT compilers known to be among the most efficient Scheme implementations and faster than other JIT compilers for Scheme. We present a detailed performance comparison in Chapter 6.

These observations, as well as the result of the experiments presented in Chapter 6 and our experience with this project indicate that the design and the techniques presented in this thesis allow building an optimizing JIT compiler with first-class performance and low development effort.

5.3. Code Transformations and Static Analyses

In addition to Gambit's GC, LC uses its frontend. We explain in Section 2.3 that we do not use any low level transformation in LC. However, when a program is executed, it is first read by Gambit's frontend that applies transformations on the AST. The resulting AST is then used as an input for LC that itself performs some analyses and transformations on this AST. In this section, we present the analyses and transformations applied by both Gambit and LC:

- **user-defined macro expansion** User-defined macro expansion is performed by Gambit's frontend directly to the AST.
- **inlining** Gambit inlines the callee function at a given call site when it statically knows its identity and if it knows that it is not mutated. The frontend does not use any advanced analysis to infer the identity of callee functions. It instead uses a naive approach to perform inlining locally to binding constructs (`let`, `let*` and `letrec`). Gambit limits code expansion using an inlining factor. LC uses Gambit's default value for this factor but allows controlling it using the `--inlining-limit` option.
- **assignment conversion** Gambit's frontend also analyses the AST to detect variables that are mutated and wraps their declarations and uses in primitives to create boxes and extract variables from boxes. Note that for these variables, LC does not track type information and does not apply any specific optimization.
- **lambda lifting** Lambda lifting is also performed by Gambit's frontend. This transformation is applied locally to binding constructs (`let`, `let*` and `letrec`). A function with free variables, that is referenced only from an application position, is lifted. It is thus possible that all the free variables are converted to arguments. However, if the function escapes, it is not lifted.
- **alpha-conversion** Once Gambit's frontend applied the transformations presented above, LC applies alpha-conversion to the resulting AST. This transformation is only used to simplify context handling in order to decrease development effort. Alpha-conversion in LC is local, meaning that it ensures that a variable has a unique name in its scope, i.e. that two variables in an environment, created by the compiler, never have the same name.

Note that no closure conversion transformation is performed. LC instead relies on a free variable analysis. This analysis is applied lazily. When a node of the AST representing a lambda expression is visited for code generation, the expression is analyzed to compute the set of free variables. The compiler uses the result of this analysis to retrieve, from the specialization context, the properties propagated for these free variables. If these properties are observed for the first time for this lambda expression, the compiler creates a new specialized entry point table for this closure creation site. The compiler also uses the result of this analysis to generate code allocating the closure and copying free variables. A limitation of the implementation of this analysis is that the compiler does not currently use a cache mechanism to reuse the result of this analysis when another version of the same closure creation site is generated.

Some compilers perform a code transformation to explicitly insert type dispatches in the AST to ensure the safety of the primitives [**54**]. LC does not apply this code transformation and instead lazily inserts, in the generated code, the type tests that are required for the type dispatches. These tests are generated only if type information is not known in the specialization context.

Finally, LC performs a liveness analysis. The compiler uses the result of this analysis to generate more efficient code by saving only the live variables to the execution stack when generating code for a call site. In its current implementation, the analysis is applied globally before executing the program. The compiler stores the result of this analysis in a hash table by storing the list of the live identifiers for each node of the AST. When a node of the AST representing a call is visited for code generation, the compiler retrieves, from this hash table, the list of variables that are lived after this call. A future work would be to completely avoid performing a liveness analysis and instead rely on BBV to only save the variables that are stored in registers that are actually used in callee functions. This approach is discussed in Chapter 7.

5.4. Compilation Contexts

The compiler uses BBV to intraprocedurally propagate constants, types and register allocation information. The types are also interprocedurally propagated. Function identities are not propagated.

```
 1: (define-record BBVContext
 2:   ;; type of each stack slot
 3:   ;;   e.g. [vec flo clo ret]
 4:   types
 5:   ;; location of each stack slot
 6:   ;;   e.g. [r0 fr0 r2 s0]
 7:   locations
 8:   ;; list of free registers
 9:   ;;   e.g. (r1 r3 r4 r5)
10:   free-regs
11:   ;; list of free machine stack slots
12:   ;;   e.g. (s1)
13:   free-slots
14:   ;; list of free floating-point registers
15:   ;;   e.g. (fr1 fr2 fr3 fr4 fr5)
16:   free-fregs
17:   ;; association list storing various properties on local variables
18:   ;;   e.g. ((d free 0 fix) (a local 2) (z local 3))
19:   environment
20:   ;; size of the machine stack frame
21:   ;;   e.g. 2
22:   frame-size
23:   ;; unique identifier of the currently generated function
24:   ;;   e.g. 84
25:   function
26: )
```

Fig. 5.1. Internal representation of a context in LC. The comments show examples of possible values for each field of the context.

In this section, we present how the compilation contexts used for code specialization are represented in LC. The implementation of the specialization contexts extends the implementation presented in Section 2.10. This observation indicates that implementing code specialization based on our compilation design does not require significant development effort.

Figure 5.1 shows how a context is represented in LC. For each property of the context, the comments in this figure show an example of possible values. LC creates, propagates and uses these contexts to specialize the code. In this section, we first explain how the properties are represented. We then explain how the contexts are handled by LC.

5.4.1. Type Property

The `types` field of `BBVContext` represents the types of the values in the current frame. In this example, there are 4 values in the frame, a vector, a flonum, a closure and an object representing a call continuation. In this case, the compiler knows all the types of the values in the frame. The two last values in the frame always are the closure of the current frame and the object representing the continuation of the current call. The list of types is managed as a stack and the first in the list is the top of the stack.

5.4.2. Register Allocation Property

The next 4 fields represent the register allocation property. `locations` is a virtual stack of the same size as the stack of types. Each slot in this stack contains the location of the corresponding value of the current frame at the same index in the type stack. In this example, the topmost value in the frame is in register `r0` and is a vector. It is possible that two slots in the type stack are associated with the same location if they represent the same value. This aliasing is important to propagate types to all the aliases.

`free-regs` and `free-slots` are free lists containing the registers and execution stack slots available to the register allocator. The machine stack pointer is updated to allocate new slots only when the register allocator needs an execution stack location and none is available in the free list. `free-fregs` is a free list containing the floating-point registers available to the register allocator.

5.4.3. Environment

The context also contains the current `environment`, represented as an association list. The key of each element is the variable symbol and the value contains various properties of the variable. The first property indicates if the variable is a local or a free variable. For a free variable, this property is followed by its index in the closure, its type and a list of its positions in context's stack. For a local variable, this property is only followed by a list of its positions in the context's stack. In this example, there is a single free variable `d` that is in the first slot of the closure. The compiler knows that `d` is a fixnum and `d` is not currently in the frame. At this point, its only location is in the closure. There also are two local variables. The variable

`a` is at slot `2` of the frame, i.e. it is a flonum that is stored in the floating-point register `fr0`. The variable `z` is at slot `3`, i.e. it is a vector that is in the general purpose register `r0`.

5.4.4. Other Information

The context also contains the current size occupied by the frame on the machine stack. In this example, two slots are allocated on the execution stack (`s0` and `s1`) so the `frame-size` field is `2`. Finally, the context contains the identifier of the currently generated function. The compiler uses this identifier to retrieve miscellaneous information on the function when generating the code such as the number of expected arguments and the use of the *rest* parameter.

5.4.5. Context Handling

The compiler must be able to apply operations on contexts. These operations add properties to a context to propagate them (e.g. the type of a new variable). They also remove properties, when values are consumed, to stop propagating them (e.g. the type of a temporary variable). Finally, they extract specific properties from contexts to help generating better code (e.g. to get the type of a variable to avoid generating a type test if the type is known). In this section, we explain how our implementation handles the contexts, that use the implementation presented in the previous sections, using relatively simple operations requiring relatively low development effort. The compiler mainly uses two operations to add and remove properties to contexts.

The context `push` operation allows adding a value on top of the frame. This operation takes the context to update, the type of the value to add, its location and an optional argument to indicate if this value is a variable or a temporary value. The compiler then creates a modified context using these arguments and returns it. The type and the location are added to the context's stacks. The free lists are updated accordingly and, if this value is associated with a variable, the environment is updated to add this slot to the variable's slot list.

The context `pop` operation removes the value on top of the frame. This operation takes a single argument that is the context. It creates an updated context and returns it. The type and the location are removed from the context's stack. The free lists are updated depending on the location that is removed. If this location is not used by other slots in the frame, it is

added to its free list. Finally, if necessary, the environment is updated. If the removed slot is the only remaining slot associated with a local variable, this variable is removed from the environment.

Other operations implicitly update contexts. That is the case for the operation returning a free register. If a register is available in the free list, this operation returns it and the context is unchanged. If no register is available, the compiler needs to spill a register. It then creates an updated context by modifying the location of the spilled frame slot and the free lists. The frame size could also be updated if no execution stack slot is available in the free list for the register spill.

All the functions returning an updated context also return a list of *moves* in addition to the new context. These moves represent the location moves that are necessary to apply the update to the current context. For example, if the operation returning a free register spills a value associated with a register and a stack slot is available in the free list, it return a list containing a single move from the register to the execution stack slot (e.g. `(r0 -> s3)`). When an updated context is returned by one of these functions, the compiler applies the moves by inserting them in the generated code.

Finally, several operations are needed to extract properties from the contexts without updating them. For example, the compiler uses an operation to return the *best* location of a variable (register or execution stack slot) and to extract the type of a variable.

5.4.6. Versioning and Version Caching

The implementation of the version caching mechanism extends the implementation presented in Section 2.8.1. To compute the key used to cache the versions, the compiler extracts from the context the properties that are used for versioning. In LC, a version is typically described by the fields `types`, `locations`, `environment` and `frame-size` of figure 5.1. However, the computation of the key may change depending on the compiler options. It is then possible that, when reusing an existing version, the compiler needs to generate code to merge the state to this version because the context used to generate it may differ in the properties that are not used for versioning.

For example, if the argument `--disable-regalloc-vers` is given to the compiler, it does not extract the register allocation properties from the context when computing a version key.

The key only contains constant and type properties. When generating a version, if a version has already been generated for this key, it is possible that it is generated for different register allocation properties. Additional instructions must then be generated to move the values to their associated locations before generating the jump to this version.

5.5. Garbage Collector

An early version of LC used a simple custom semi-space stop-and-copy GC based on Cheney's algorithm [**35**] and written in Scheme. A major disadvantage of using this approach in LC is that, because LC is itself written in Scheme, two GCs are used at run time. The first is the GC LC uses to collect the objects created by the execution of the source program and the second is the GC Gambit uses to collect the objects created by the execution of LC's compiler (i.e. the objects created to JIT compile the source program).

We later decided to integrate the objects allocated by LC to Gambit's heap to avoid using multiple GCs. Because the objects created by LC use a representation mostly compatible with the objects created by Gambit, this integration required low effort. This approach allows using a single GC when executing a source program with LC but increases the average GC pause time. Gambit's GC uses both a Mark-and-Sweep [**77**] approach for big objects and a semi-space stop-and-copy approach for small objects.

A disadvantage of this approach is that because the heap stores the objects created by both Gambit and LC, more objects are copied when the GC is triggered, impacting its performance. Each time the code generated by LC triggers the GC, the live objects of LC (created by Gambit) must also be copied. If two separate heaps were used, only the objects created by the code generated by LC would need to be copied.

However, a significant advantage of this approach is that because both compilers share the same object representation and the same heap, the objects created by one can be used by the other. This allowed increasing the set of features LC offers with almost no effort by allowing calling Gambit code from the code executed by LC.

Moreover, because there are existing mature GCs, we had no particular reason to create one from scratch. Reusing the GC of Gambit allowed us to save development effort.

For our research, we aimed to use existing components of Gambit without requiring to fork its code and maintain our own version. We ended up forking Gambit but only to add support of NaN-boxing to its GC, which Gambit does not use.

Our experience with GCs suggest that, when building a JIT compiler, a trade-off between the number of reused existing components (i.e. the development effort) and the number of software dependencies (i.e. the control over the components) must be made.

When minimizing development effort is more important than maximizing control, integrating to an existing system is a wise choice. In our case, it allowed using a mature GC with no significant effort.

When maximizing control is more important than minimizing development effort, building custom components is interesting. To mitigate the development effort increase, simple techniques can be used but performance will be impacted. As an example, before integrating to Gambit's GC, we implemented our simple stop-and-copy GC in less than 400 lines of Scheme code.

To implement eager unboxing and solve the issue mentioned in Section 4.6 (allow the GC to collect objects if both boxed and unboxed objects are stored in the stack), we initially used two separate stacks to store boxed and unboxed values. To eliminate the negative impact on performance of this approach, we decided to use the solution based on stack maps. We did not use the conservative approach because Gambit's GC uses a precise copying strategy.

5.6. Gambit Bridge

To take advantage of the fact that the objects are shared by LC and Gambit, we have implemented a simple bridge allowing calls to Gambit functions (including the code of LC) from the code generated by LC. The bridge allows programmers using LC to write more complex applications because the libraries available in Gambit can be used without requiring an LC specific implementation.

To import Gambit functions to the code executed by LC, each call using the bridge must use the `gambit$$` prefix in front of the Gambit global variable bound to the function.

Figure 5.2 shows an example of using the Gambit bridge to use features not implemented directly by LC. The bridge calls `shell-command`, `getenv`, `file-exists?` and `println` are

```
1:  ; Declare functions that use the bridge
2:  (define (shell-command cmd) (gambit$$shell-command cmd))
3:  (define (getenv var)        (gambit$$getenv var))
4:  (define (file-exists? path) (gambit$$file-exists? path))
5:  (define (println value)     (gambit$$println value))
6:
7:  ; We can then use them as regular Scheme functions
8:
9:  ; Execute the shell command "ls"
10: (shell-command "ls")
11:
12: ; Print the value of the environment variable $HOME
13: (println (getenv "HOME"))
14:
15: ; Check if a file exists and print boolean result
16: (println (file-exists? "test.scm"))
```

Fig. 5.2. Sample of use of the bridge to access the host environment using non-standard Scheme functions available in Gambit.

wrapped in Scheme functions defined in the global scope. These functions can then be used as regular Scheme functions to access the host environment.

Functions of the standard library of LC can be implemented using the bridge. However, a bridge call is significantly slower than a call to a function natively implemented in LC. For this reason, only I/O functions of the LC standard library, that are non performance-critical, are implemented through this bridge.

5.6.1. Implementation

In LC, a call with an operator using the bridge prefix is considered as a special form. When the compiler generates the code for one of these special forms, it first evaluates the arguments. It then inserts a call to itself in the generated code, using the same implementation as for the compilation stubs. The arguments of this call are the prefixed symbol and the evaluated arguments.

When the code generated for this call is executed, LC is called, it retrieves the symbol representing the operator, evaluates it to get the Gambit function bounded to it and calls this function using the other arguments. Once the call returns, LC writes the returned value

to the execution stack and the control flows back to the code generated by LC that retrieves the returned value from the stack and writes it to its allocated register.

Note that a value returned by a call using the bridge initially has an unknown type in LC.

5.7. Tools

We have built LC specific tools to help us debug and analyze the impact of the techniques on various metrics. Some tools also facilitate data rendering for the papers and this dissertation and help improve the reproducibility of the experiments. In particular, we used them to build the software artifact for our ECOOP paper.

5.7.1. Data Extraction

Four command line options can be given to the compiler to measure various properties of the execution of a program (`--stats`, `--stats-full`, `--time` and `--ctime`).

The `--stats` option allows generating a report displaying the properties for which the collection does not significantly impact performance. This report is exported as a list of key-value pairs in the Comma-Separated Values (CSV) format to facilitate automated data extraction. Figure 5.3 shows an example of a report generated with the `--stats` option for an execution with a version limit set to 5. We see that the compiler displays the metrics that are analyzed and presented in Chapter 6, including the number of executed boxing and unboxing operations and the number of executed type tests.

Collecting some metrics significantly impact performance. The `--stats-full` option collects the metrics significantly impacting performance and the ones collected with the `--stats` option and displays a report. For example, using this option, all the function calls are traced and the number of arguments used at call sites is counted.

The `--time` options measures the execution time and the `--ctime` option measures the compilation time.

5.7.2. Benchmarking

We built a generic benchmarking environment allowing measuring the execution properties for various Scheme implementations. This tool automates the benchmarks execution and implements and uses the methodology of the experiments presented in the next chapter. This

```
./lc program.scm --max-versions 5 --stats
Closures: 653005
Executed tests: 135338024
Flonum boxing operations: 7
Flonum unboxing operations: 11
Bytes allocated: 300576560
Code size (bytes): 37870
Stub size (bytes): 10896
Total size (bytes): 48766
Global cc table size: 13
Global cr table size: 8
Number of cctables: 137
Number of crtables: 119
CC table space (bytes): 14248
CR table space (bytes): 15232
Min versions number: 0
Max versions number: 5
```

Fig. 5.3. Example of using the `--stats` option of LC to collect execution information of a Scheme program.

tool generates a report displaying the metrics, for all the implementations, in the CSV format to facilitate automated data extraction.

5.7.3. Data Visualization

We also built data visualization tools to help debug and visualize the behavior of LC to evaluate the techniques. One of them generates graphs from the reports generated by the benchmarking environment tool and from the data extraction tools. For our project, we used this tool to help understand the impact of the techniques and optimizations, for non-regression testing when implementing techniques and to render data for the papers. We also used this tool to generate the graphs presented in the next chapter.

Using a compiler based on interprocedural code specialization raises issues that do not typically exist in other implementations or have less impact. These issues include the possible explosion in the number of versions and possible wrong compilation context handling of the compiler that is bug-prone. In particular, a bug in context handling can cause the propagation of invalid types causing the generation of versions specialized for the wrong types. In this situation, an error can be raised later in the execution and determining the source of the bug

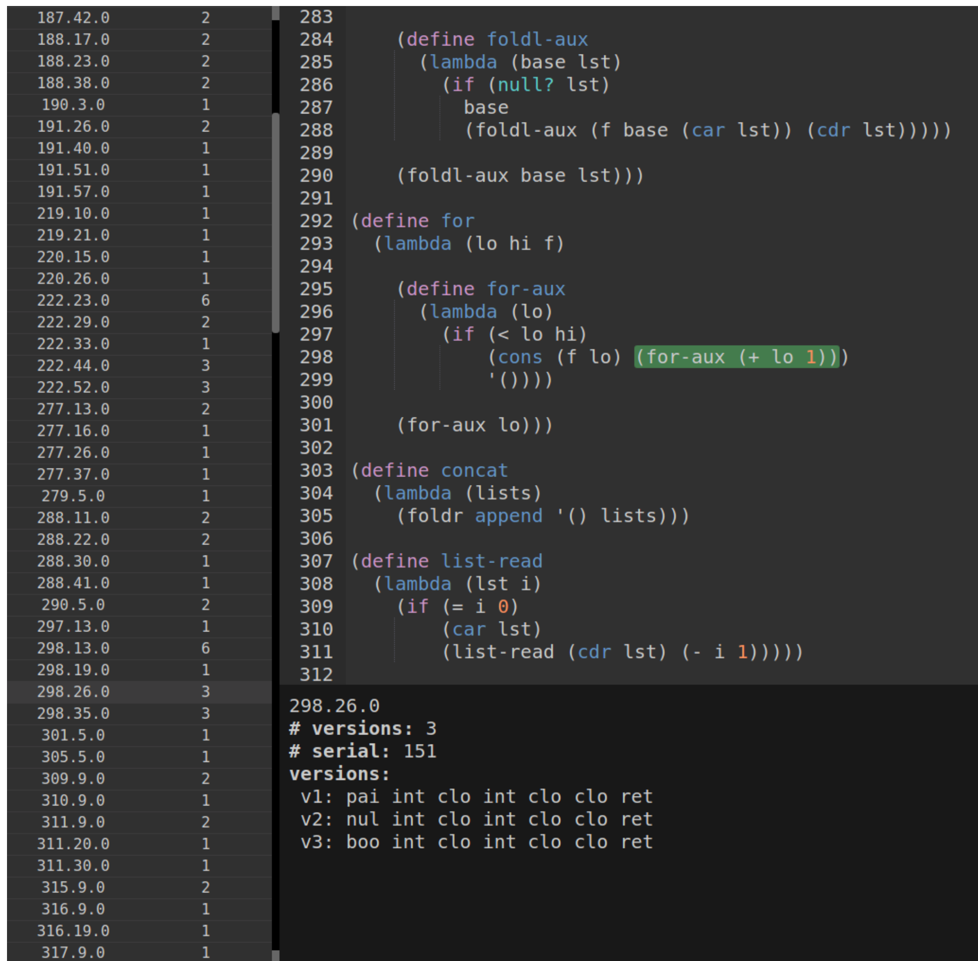


Fig. 5.4. Visualization of the versions generated for a program from a report generated by LC.

may be difficult. To help us solve these issues, we built a tool using the reports generated after executing a benchmark to visualize, in a user-friendly way, the versions generated by the compiler.

Figure 5.4 shows an example of use of this tool. The user can select any sub-expression from the executed program to visualize the number of versions generated for this sub-expression as well as the contexts used to generate these versions. For the expression currently selected, three versions are generated. The first version is specialized for a frame containing a pair (`pai`), a fixnum (`int`), a closure (`clo`), another fixnum, two closures and an object representing the continuation of the current call (`ret`). These types are the types of the temporary value `(f lo)`, the local variable `lo` and the free variables `f`, `hi` and `for-aux`.

This tool helped us debug LC and analyze how the compiler specializes programs. In particular, it allowed us to find and explain sources of overspecialization and helped us tweak the techniques to generate code that is as efficient as possible.

5.8. ECOOP Artifact

We packaged LC and these tools in an artifact [**115**] that is published with our ECOOP paper presenting the interprocedural extensions of BBV [**114**].

This artifact contains a virtual appliance, packaged using the Open Virtualization Format (OVF), and its documentation. This format improves the portability of the artifact by allowing its execution through a system virtual machine.

It contains the build of LC we used for the experiments presented in the paper and it allows interested parties to easily extract the data presented in the paper such as the number of executed type tests or the generated code size. This artifact makes it easy to repeat the experiments using a set of standard benchmarks as well as new programs.

5.9. Summary

The design we present in Chapter 2 can be implemented with relatively low development effort in an optimizing JIT compiler for a large subset of the Scheme programming language. The compiler can use a simple representation of the specialization contexts and several simple operations on them. Our implementation of this design for a real JIT compiler is relatively close to the implementation for the small abstract language presented in Chapter 2.

Our project suggests that implementing the interprocedural extensions and eager unboxing does not require a significant effort either. Indeed, LC is able to generate optimized code whereas it is developed by a single person and weighs in at 20K lines of code.

Development effort can further be decreased by reusing components of existing implementations as we have done with the Gambit Scheme system.

Chapter 6

Results

To support and encourage reproducibility of the experiments, we published an artifact with our ECOOP 2017 paper:

- Baptiste Saleil and Marc Feeley. Interprocedural Specialization of Higher-Order Dynamic Languages without Static Analysis (Artifact). In *31st European Conference on Object-Oriented Programming, ECOOP, 2017.* [**114**]

In this chapter, we present the experiments we conducted with LC. These experiments aim to validate our approach by showing that a system built with relatively low development effort can achieve good performance.

We explain in Chapter 5 how we built LC with relatively low development effort. In this chapter, we analyze the impact of our design and techniques on the generated code in order to evaluate its impact on the overall performance, but also to ensure that our approach does not significantly impact the compilation (in particular the compilation time and the generated code size).

We evaluate the quality of the code generated by our techniques by measuring the number of executed type tests and the number of executed boxing and unboxing operations. We then measure the impact on the execution and compilation time. To evaluate the overall performance that a system built with low development effort can achieve, we also compare the performance of LC with mature AOT and JIT Scheme implementations.

6.1. Benchmarks

For these experiments, we used the benchmarks distributed with the Gambit Scheme compiler. This set of benchmarks is a standard Scheme benchmark suite, often used to evaluate Scheme implementations. Not all the benchmarks are included in our experiments because some use features not supported by LC such as first-class continuations. The number of iterations within each benchmark is also taken from Gambit and is chosen so that each benchmark is executed in a short but reliably measurable time, typically on the order of 0.25 to 1 second.

We used 38 benchmarks for our experiments. Some of them are micro-benchmarks used to test a specific feature of the language (`array1`, `diviter`, `divrec`, `sum`, `sumfp` and `sumloop`). Several benchmarks implement classical simple recursive functions (`ack`, `fib` and `tak`) as well as variants of these functions (`cpstak`, `fibfp` and `takl`). The other benchmarks are bigger and implement more complex algorithms. 9 benchmarks use floating point numbers intensively (`fft`, `fftrad4`, `fibfp`, `mbrot`, `nbody`, `nucleic`, `pnpoly`, `simplex` and `sumfp`). We used these benchmarks to evaluate the impact of flonum eager unboxing. `compiler` is the biggest benchmark of the suite (≈12K lines of code). Because it is the biggest and it uses values of a variety of types, it is a good representative of the typical use-case for Scheme.

6.1.1. Methodology

We collect various metrics for each benchmark. For non-time metrics such as the number of executed type tests, the results are deterministic. For these metrics, the benchmarks are executed a single time and data are extracted from this execution. For time metrics (execution time, compilation time and GC time), each benchmark is executed 10 times. A new process is started for each execution, the minimum and maximum times are removed and the arithmetic mean of the 8 remaining times is used for the evaluation. With our setup, we observed a relative standard deviation that is typically below 1% for these 8 remaining times.

For some of the experiments, we compare the results to LC using a naive compilation approach. When using naive compilation, LC does not use BBV, only generic basic block versions are generated and the code is not specialized. This can be considered as a naive code generation pass from ASTs to machine code, without optimization.

To evaluate the performance of the code generated by LC, we compare its execution time to the execution time of the code generated by two mature AOT Scheme implementations, namely Gambit [**51**] and Chez [**135**]. We compare the execution time with these two implementations in both *safe* and *unsafe* mode. In safe mode, the compiler inserts checks to prevent crashes at run time. These checks typically include vector bounds checking, stack overflow checking and type checking for primitives and calls. However, in safe mode, the compiler still uses generic arithmetic (it inserts type dispatches for arithmetic operators). LC offers a single mode that inserts part of these checks, it does not perform vector bound checking nor stack overflow checking. Consequently, the single mode of LC can be considered as being between the safe and unsafe mode of the Gambit and Chez compilers. We show in Section 6.5.1 that LC generates code that competes with these implementations.

6.1.2. Setup

The experiments were conducted on an otherwise unused machine using an Intel Core i7-7700K 4.20GHz CPU, with 16GB DDR4 RAM running the GNU/Linux operating system.

The LC results have been collected from the benchmarks executed with a hard limit in the number of specialized versions set to 5. Previous experiments [**37**] showed that a limit of 5 is rarely reached for JavaScript programs. Our experiments indicate that, in most cases, there is no significant difference for Scheme programs. In the rest of this chapter, we mention the cases in which this limit impacts the results.

6.2. Intraprocedural Basic Block Versioning and Compilation Design

In this section, we evaluate the implementation of intraprocedural BBV in LC. The interprocedural extensions are evaluated in the next section. These experiments first allow us to evaluate the impact of BBV on Scheme programs and to validate the implementation of the design, presented in Chapter 2, in a real JIT compiler. Because the code is specialized for types to remove type tests, it is interesting to analyze the impact on the number of executed type tests. We define a type test as an operation testing if a value is of a given type. The type tests include the checks required to ensure the safety of the primitives (e.g. the primitive `car` testing for `pair?`), type dispatches for polymorphic primitives (e.g. `+`) and the type

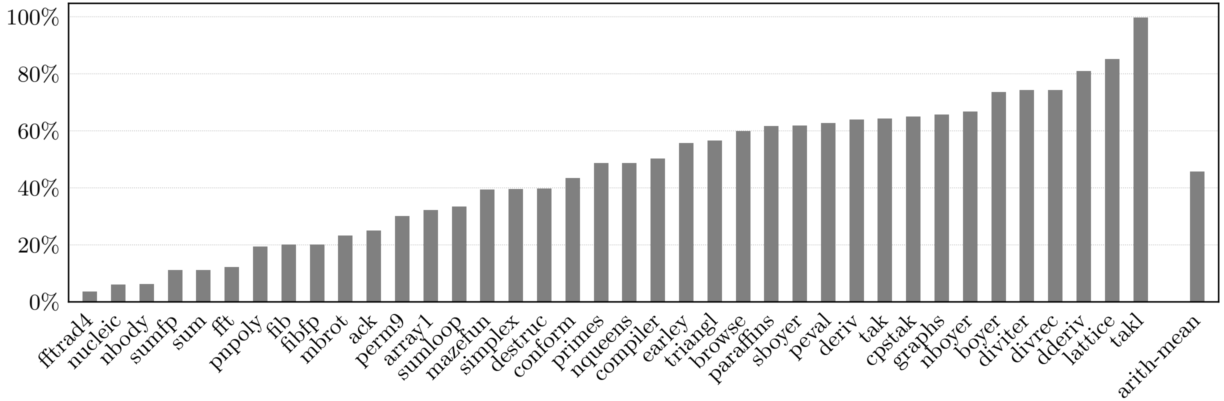


Fig. 6.1. Proportion of remaining executed tests using intraprocedural BBV relative to naive compilation for LC with a version limit of 5 (naive compilation is 100%).

predicates of the Scheme language (e.g. `pair?`). In the case of polymorphic primitives, only the tests executed to determine the type of an argument are counted. For example, if the primitive `+` is called with a value of unknown type, two tests are executed for the dispatch if the value is a flonum (because LC first tests if a value is a fixnum, then tests if it is a flonum). We also analyze the impact on the size, the execution time and the compilation time of the generated code.

6.2.1. Number of Type Tests

To evaluate the impact of BBV on the generated code, Figure 6.1 shows the number of remaining executed type tests for each benchmark executed with LC using intraprocedural BBV relative to LC using naive compilation (naive compilation is 100%). We see in the figure that for all benchmarks excepted `takl`, a significant number of tests are removed. `fftrad4` is the most positively impacted. Using intraprocedural BBV, 96.5% of the tests are removed for this benchmark.

`takl` is the least impacted benchmark, almost no test is removed. The reason is that this micro-benchmark only uses lists for its computation. Figure 6.2 shows the code of the `takl` benchmark. The main computation is in the `mas` and `shorterp` functions. This benchmark implements the Takeuchi function [**90**] using lists as numbers. The only operation executed on values is the `cdr` operation. Because the compiler does not track the type of values in pairs, each element retrieved with `cdr` has an unknown type. Consequently, the compiler

```
 1: (define (listn n)
 2:   (if (= n 0)
 3:       '()
 4:       (cons n (listn (- n 1)))))
 5:
 6: (define l18 (listn 18))
 7: (define l12 (listn 12))
 8: (define  l6 (listn 6))
 9:
10: (define (mas x y z)
11:   (if (not (shorterp y x))
12:       z
13:       (mas (mas (cdr x) y z)
14:            (mas (cdr y) z x)
15:            (mas (cdr z) x y))))
16:
17: (define (shorterp x y)
18:   (and (not (null? y))
19:        (or (null? x)
20:            (shorterp (cdr x)
21:                      (cdr y)))))
22:
23: (mas l18 l12 l6)
```

Fig. 6.2. Scheme code of the `takl` benchmark. This benchmark implements the Takeuchi function using lists as counters.

does not know that it is a pair and must insert a type check for each `cdr` operation to check that the argument is a pair.

Intraprocedural BBV allows removing 54% of the type tests on average. This figure shows that when applied to Scheme, BBV significantly reduces the number of type tests executed.

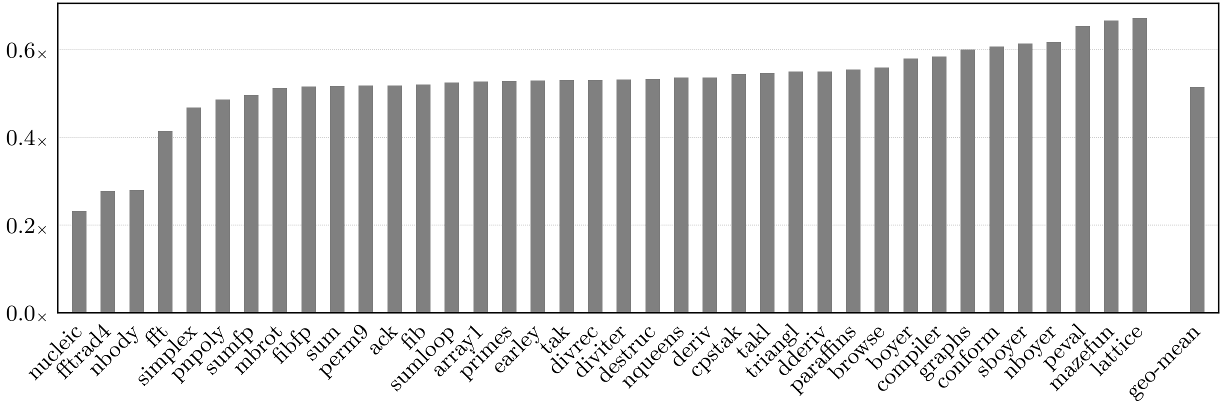


Fig. 6.3. Size of the generated code using intraprocedural BBV relative to naive compilation for LC with a version limit of 5 (naive compilation is 1.0).

6.2.2. Generated Code Size

To show that our design does not significantly impact the code size, Figure 6.3 shows the size of the code generated by LC using intraprocedural BBV relative to the size of the code generated by LC using naive compilation (naive compilation is 1.0). Because the generated code is more optimized, we observe a decrease in the code size on average. Flonum intensive benchmarks are the most affected because using BBV and eager unboxing, many type tests and boxing/unboxing operations are removed resulting in a significant decrease in the size of the generated code compared to naive compilation. The most affected benchmark is `nucleic` with a code that is 0.23× naive compilation. The least impacted is the `lattice` benchmark. For this benchmark, the code is smaller by a factor of 0.67× . On average, the generated code is smaller by a factor of 0.51× . This is due to the fact that BBV allows generating fewer type tests and fewer boxing and unboxing operations and it is a consequence of the design LC uses. LC relies on BBV to remove type tests, to propagate constants, to unbox flonums and to optimize register allocation so using naive compilation, these optimizations are not applied and more code is generated.

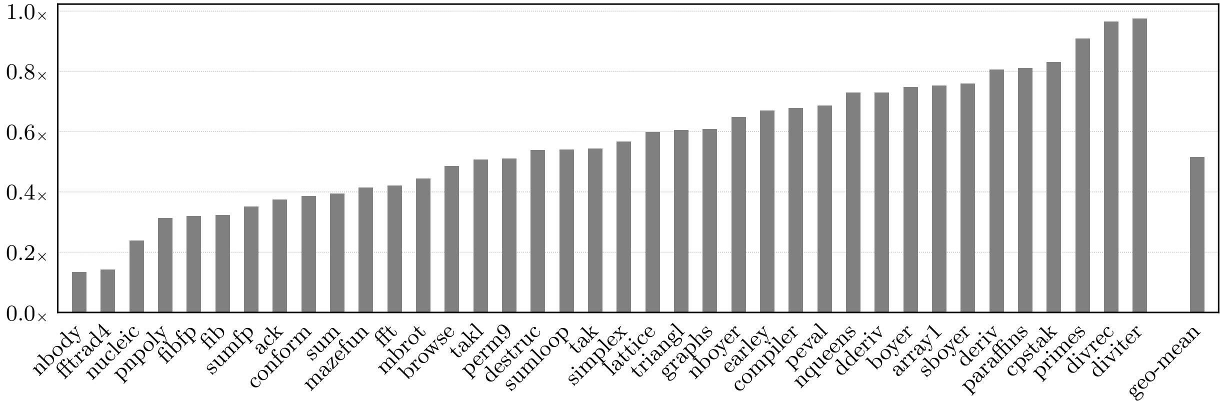


Fig. 6.4. Execution time of the generated code using intraprocedural BBV relative to naive compilation for LC with a version limit of 5 (naive compilation is 1.0).

6.2.3. Execution Time

To evaluate the impact of intraprocedural BBV on performance, Figure 6.4 shows the execution time of the code generated by LC using intraprocedural BBV relative to the execution time of the code generated by LC using naive compilation (naive compilation is 1.0). For this experiment, the time includes the execution time and excludes the GC time and the compilation time. We see in the figure that all the benchmarks run faster with intraprocedural BBV. For the same reasons as for the code size, the flonum intensive benchmarks are the most affected. The best case is `nbody`. Its execution time using intraprocedural BBV is smaller by a factor of 0.13× . The least affected benchmark is `diviter` that is slightly faster with intraprocedural BBV. Its execution time using intraprocedural BBV is 0.97× the execution time of naive compilation. Because our implementation does not track the type of compound data types, the benchmarks affected least are the ones intensively using compound data types. For instance, the benchmarks `divrec` and `diviter` intensively use pairs thus the type properties discovered by BBV are quickly lost when creating new pairs resulting in a code that is not significantly optimized. In Section 7.1, we briefly explain how this issue could be addressed. The benchmarks intensively using higher-order functions are also minimally affected. For instance, BBV allows discovering types for the `cpstak` benchmark, but because it is written in CPS, the types are quickly lost in the absence of interprocedural specialization. Overall, all the benchmarks except `divrec` and `diviter` are significantly

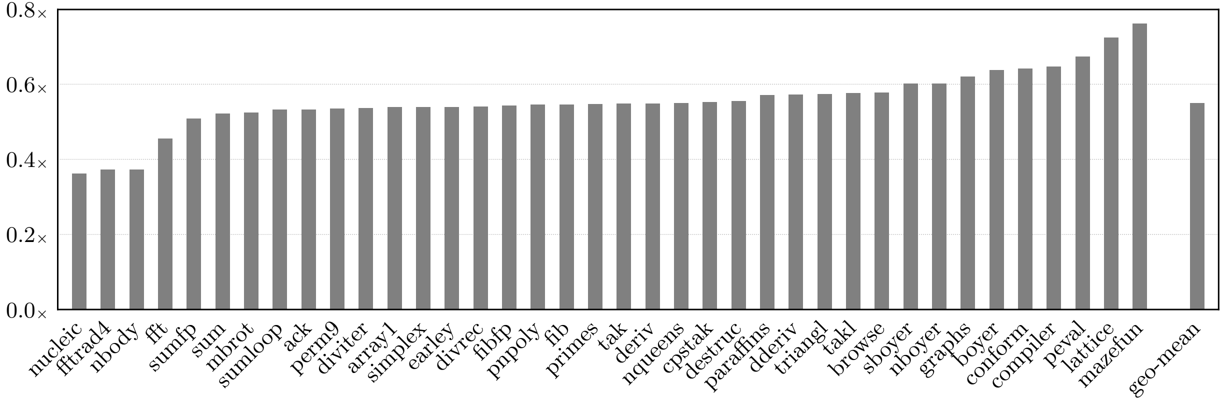


Fig. 6.5. Compilation time of the generated code using intraprocedural BBV relative to naive compilation for LC with a version limit of 5 (naive compilation is 1.0).

faster when intraprocedural BBV is used compared to a naive compilation from the AST. On average, the benchmarks are faster. The execution time is 0.52× naive compilation.

6.2.4. Compilation Time

To show that our approach does not significantly impact the compilation time, Figure 6.5 shows the compilation time of the benchmarks executed by LC using intraprocedural BBV relative to the compilation time of the benchmarks executed by LC using naive compilation (naive compilation is 1.0). Because less machine code is generated, the compilation time is significantly decreased for all the benchmarks. Flonum intensive benchmarks are again the most affected, because many operations are removed. For the `nucleic` benchmark, the most impacted, the compilation time using intraprocedural BBV is 0.36× naive compilation. For the `mazefun` benchmark, the least impacted, the compilation time using intraprocedural BBV is still 0.76× naive compilation. This experiment shows that an optimizing compiler, using BBV and the design we presented, generates code that is significantly optimized compared to a naive compilation from the AST resulting in a significant decrease of the compilation time. The compilation time using intraprocedural BBV is 0.55× naive compilation on average.

6.2.5. Summary

These experiments first show that in the context of Scheme, intraprocedural BBV allows generating code that is significantly more efficient than the code naively generated from an AST. They also validate the design presented in Chapter 2. Overall, our implementation

allows to significantly decrease both the execution time and the compilation time of all the benchmarks. Moreover, because our implementation of eager unboxing removes flonum boxing operation, the GC time of the flonum intensive benchmarks is also significantly decreased. The code generated for Scheme programs is more optimized and the overall performance is improved.

6.3. Interprocedural Extensions

In this section, we repeat the analysis of the previous section but with the interprocedural extensions presented in Chapter 3. LC does not use a specific strategy to limit the size of the entry point tables. It creates tables using a global layout that is big enough to store all the contexts, used to specialize the entry points, that are observed when the program is executed. In this section, we also analyze the memory footprint of the function and continuation entry point tables.

6.3.1. Number of Type Tests

Figure 6.6 shows the number of executed type tests that remain for each benchmark executed with LC using interprocedural BBV relative to LC using intraprocedural BBV (intraprocedural BBV is 100%). We see that for 12 benchmarks, all the remaining tests are removed. The interprocedural extensions allow removing almost all tests for most of the flonum intensive benchmarks. However, the benchmarks intensively using compound data types such as `divrec` and `diviter` are not significantly affected because the compiler does not track compound types thus both intraprocedural and interprocedural BBV are not effective at removing tests. In the best case, all the tests are removed for 11 benchmarks. In the worst case, for 4 benchmarks, no additional test is removed using the interprocedural extensions. A significant number of tests are removed for the other benchmarks. On average, 52% of the tests that intraprocedural BBV is not able to remove are eliminated using interprocedural BBV.

Figure 6.7 shows the number of remaining executed type tests for each benchmark executed with LC using interprocedural BBV relative to LC using naive compilation (naive compilation is 100%). The graph also shows the number of executed type tests with LC using intraprocedural BBV in light gray for reference. Using interprocedural BBV, almost

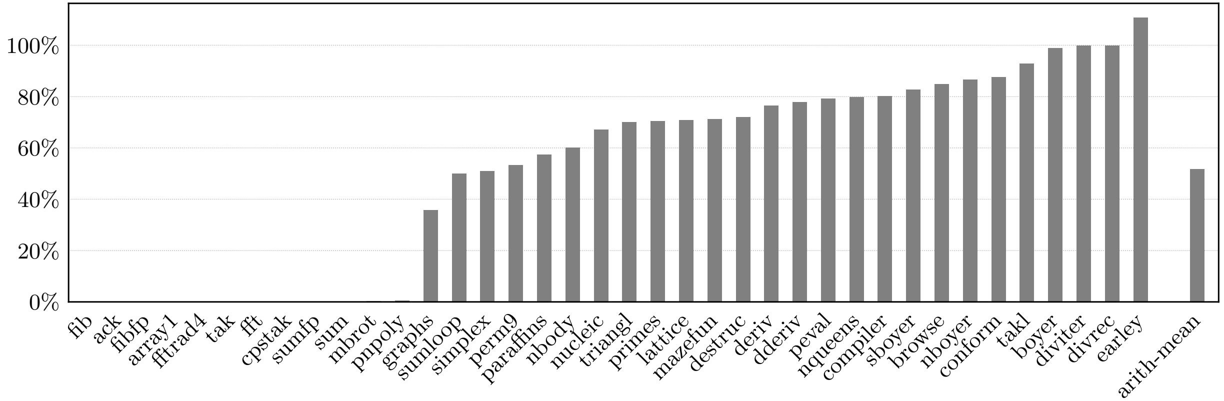


Fig. 6.6. Proportion of remaining executed tests using interprocedural BBV relative to intraprocedural BBV for LC with a version limit of 5 (intraprocedural BBV is 100%).

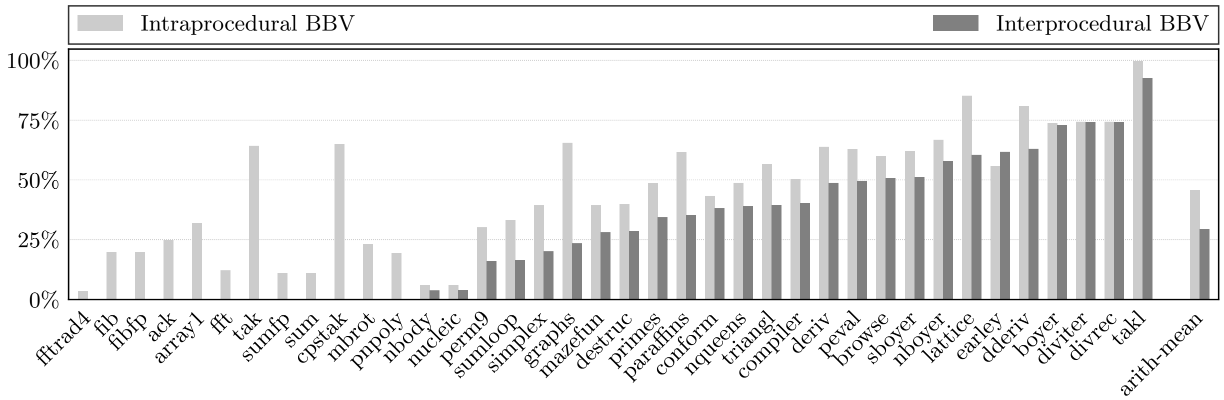


Fig. 6.7. Proportion of remaining executed tests using intraprocedural and interprocedural BBV relative to naive compilation for LC with a version limit of 5 (naive compilation is 100%).

all tests required by naive compilation are removed for 14 benchmarks. All the benchmarks execute significantly fewer tests compared to naive compilation. The worst case is the `takl` benchmark, only 7.4% of the tests required by naive compilation are removed.

On average, 70% of the tests are removed with interprocedural BBV. These experiments show that the interprocedural extensions allow collecting significantly more dynamic properties than intraprocedural BBV. Using these properties, the compiler is able to remove more dynamic operations and generate more optimized code.

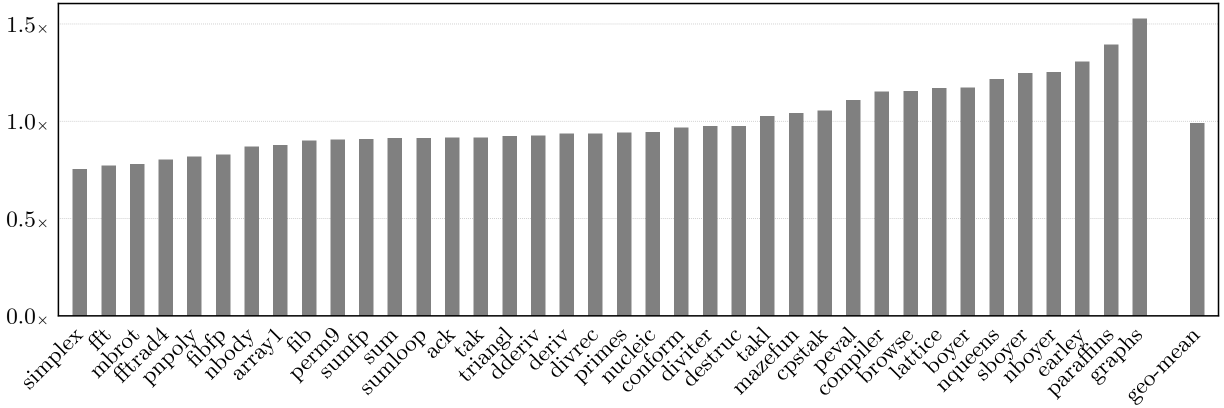


Fig. 6.8. Size of the generated code using interprocedural BBV relative to intraprocedural BBV for LC with a version limit of 5 (intraprocedural BBV is 1.0).

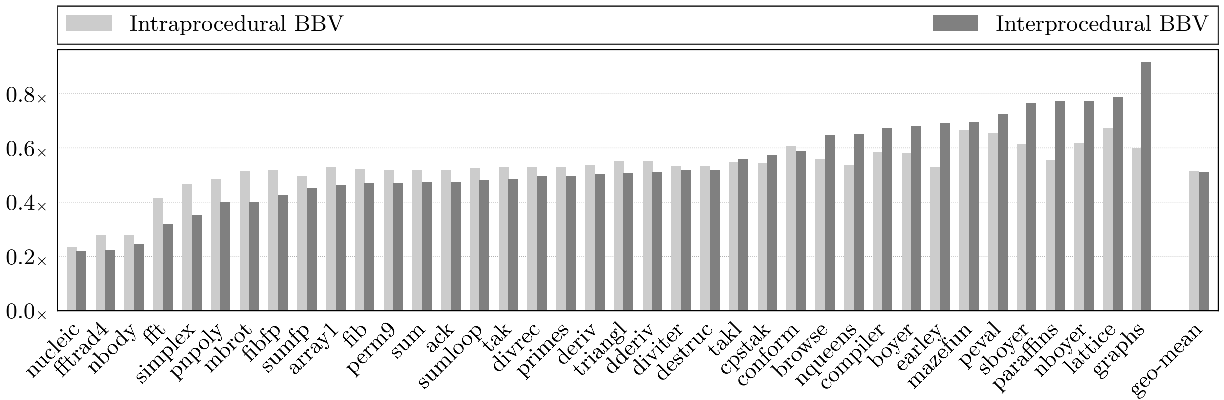


Fig. 6.9. Size of the generated code using intraprocedural and interprocedural BBV relative to naive compilation for LC with a version limit of 5 (naive compilation is 1.0).

6.3.2. Generated Code Size

Figure 6.8 shows the size of the code generated by LC using interprocedural code specialization relative to intraprocedural code specialization for LC with a version limit of 5 (intraprocedural BBV is 1.0). The impact on the code size depends on the nature of the benchmark. For flonum intensive benchmarks, less code is generated because, in addition to the type tests, a significant number of flonum boxing and unboxing operations are removed by eager unboxing. For more generic benchmarks using values of more types, we instead observe a moderate increase in the code size. Using interprocedural code specialization, the compiler collects more information than when using intraprocedural specialization. The generated

code contains fewer operations but, because more types are collected, more versions are generated for these benchmarks, resulting in an increase in the code size. For most of the benchmarks, the size of the generated code is not significantly affected. The biggest decrease occurs for the `simplex` benchmark. For this benchmark, the code is smaller by a factor of 0.76× . The biggest increase occurs for the `graphs` benchmark, with a factor of 1.53× . The size of the code generated using interprocedural code specialization is almost the same as intraprocedural code specialization on average.

Figure 6.9 shows the size of the code generated by LC using interprocedural specialization relative to naive compilation (naive compilation is 1.0). The graph also shows the size of the code generated by LC using intraprocedural specialization in light gray for reference.

We see in the figure that the increase in the code size we observed for some benchmarks is mitigated by the decrease we observed with intraprocedural specialization. Compared to naive compilation, there is a significant decrease in the code size for all the benchmarks, including `graphs`, the benchmark that showed the most significant increase compared to intraprocedural specialization. For this benchmark, the size of the code generated using interprocedural specialization is 0.92× the size of the code generated by naive compilation. The most affected benchmark is `nucleic`. For this benchmark, the size of the code generated using interprocedural specialization is 0.22× naive compilation. Overall, the generated code using the interprocedural extensions is significantly smaller than the code generated by naive compilation. On average the code is 0.51× the size of the code generated by naive compilation.

These experiments show that the increase in the number of properties discovered by interprocedural BBV causes an increase in the code size compared to intraprocedural BBV. However, this increase is mitigated by the decrease we observed with intraprocedural BBV.

6.3.3. Function and Continuation Tables Size

Table 6.1 shows the amount of memory used by the function and continuation entry point tables when executed with interprocedural specialization. The value shown in the table is the minimum amount required to store in the global layout the slots of all the contexts used to specialize the entry points at execution. We assume the compiler does not use a fallback strategy nor a fixed limit in the size of the global layout. Depending on the strategy the compiler uses to handle table overflows, such as those presented in Section 3.5.2, this amount

Benchmark	Function tables (kB)	Number of contexts	Continuation tables (kB)	Code size (kB)
compiler	876	62	396	611
conform	24	16	23	52
mazefun	23	21	16	42
sboyer	18	10	18	48
nboyer	18	10	19	49
graphs	18	13	14	56
nucleic	17	7	18	69
peval	15	10	32	78
earley	8	5	17	69
others	< 7	< 7	< 13	< 47

Tab. 6.1. Memory used by the function and continuation entry point tables when using interprocedural code specialization. This table also shows the number of contexts used for function entry point specialization and the generated code size.

may vary. If the compiler uses a fallback strategy to automatically fallback to existing similar contexts to limit the size of the global layout, the tables may use less memory. If the compiler uses a dynamic resizing approach using a growth factor, the tables may use more memory. For the continuation tables, we assume the compiler uses a fixed layout.

We see that for the micro-benchmarks, the tables do not use a significant amount of memory. For more complex benchmarks using values of different types (the benchmarks shown in the table), more contexts are used to specialize the functions and the continuations thus more memory is used by the tables. For the benchmark `compiler`, 876kB are used by the function entry point tables and 396kB are used by the continuation entry point tables. Compared to the size of the generated code, the function and continuation entry point tables use less memory, except for `compiler`. For this benchmark, they use about 2× the memory of the generated code. However, the size of the tables and the size of the generated code do not exceed 2000kB which is not significant for current systems. Although the amount of memory used by the tables is not significant for current systems, it has to be considered if memory usage is an important concern (e.g. for embedded systems).

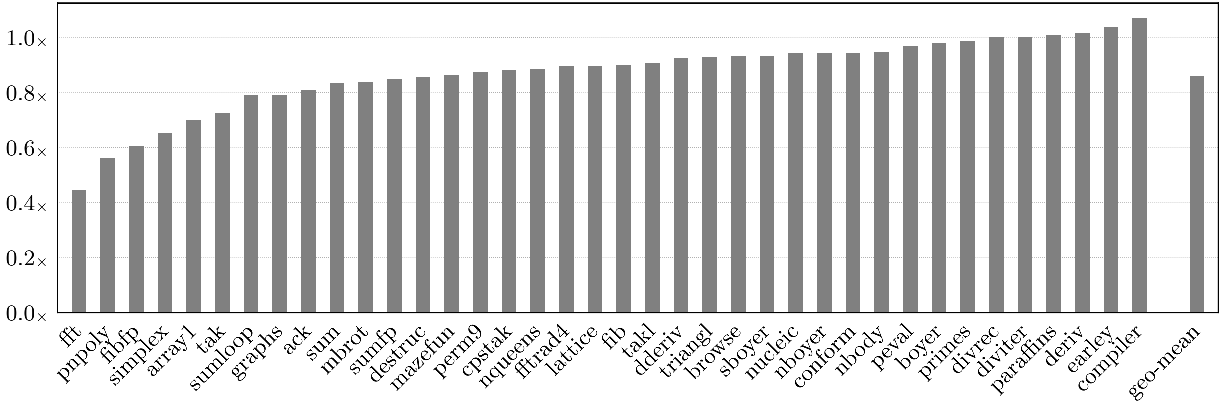


Fig. 6.10. Execution time of the generated code using interprocedural BBV relative to intraprocedural BBV for LC with a version limit of 5 (intraprocedural BBV is 1.0).

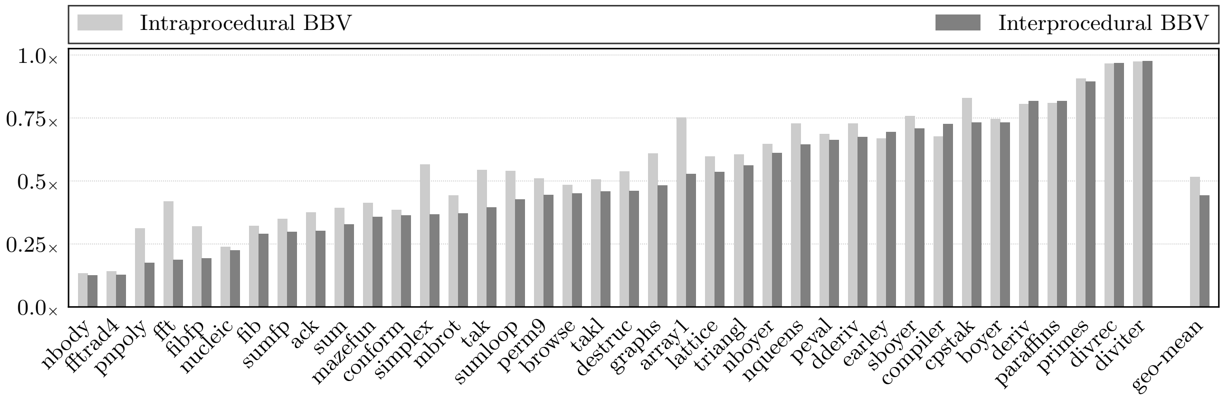


Fig. 6.11. Execution time of the generated code using intraprocedural BBV and interprocedural BBV relative to naive compilation for LC with a version limit of 5 (naive compilation is 1.0).

6.3.4. Execution Time

Figure 6.10 shows the execution time of the code generated by LC using interprocedural code specialization relative to intraprocedural code specialization (intraprocedural specialization is 1.0). Because interprocedural specialization allows removing more operations, the code generated is significantly faster for most of the benchmarks. The `fft` benchmark is the best case. The execution time of the code generated for this benchmark using interprocedural specialization is 0.45× intraprocedural specialization. The `compiler` benchmark is the worst case. The execution time is 1.07× intraprocedural specialization. The slowdown observed for

this benchmark is due to the fact that it intensively uses compound data types and intensively uses small and frequently called functions. The negative impact of the interprocedural extensions on the generated code (in particular the indirection required at function calls) is more significant than the positive impact the extensions have, even if they allow removing more tests for this benchmark. The `compiler` benchmark is an early version of the Gambit Scheme compiler and like other compiler benchmarks, such as the `gcc` benchmark of the SPEC suite[1], it is difficult to optimize by nature.

This figure shows that the extensions are especially efficient at generating optimized code for programs doing intensive numerical computations (e.g. `fft` and `mbrot`) and exploiting a functional programming style (e.g. `cpstak` and `mazefun`). However, the extensions are less efficient for benchmarks intensively using compound data types and small functions (e.g. `compiler`, `diviter` and `divrec`).

Overall, a speedup is observed when using interprocedural code specialization. On average, the execution time is 0.86× intraprocedural BBV.

Figure 6.11 shows the execution time of the code generated by LC using interprocedural specialization relative to the execution time of the code generated by LC using naive compilation (naive compilation is 1.0). The figure also shows the execution time of the code generated by LC using intraprocedural specialization in light gray for reference. We explained that for two benchmarks (`divrec` and `diviter`), both intraprocedural and interprocedural specialization have no effect on the performance of the generated code. However, we see in this figure that with interprocedural BBV, the compiler generates code that is significantly faster than naive compilation for all of the other benchmarks.

The `nbody` benchmark is the best case. The execution time of the code generated for this benchmark using interprocedural specialization is 0.13× naive compilation. The `diviter` benchmark is the worst case. The execution time is 0.98× naive compilation.

Overall, a significant speedup is observed when using interprocedural code specialization compared to naive compilation. The execution time using interprocedural specialization is 0.44× naive compilation on average.

Similarly to intraprocedural specialization, limiting the number of versions to 5 does not significantly impact the performance of the generated code because this limit is rarely reached.

[1]https://www.spec.org/

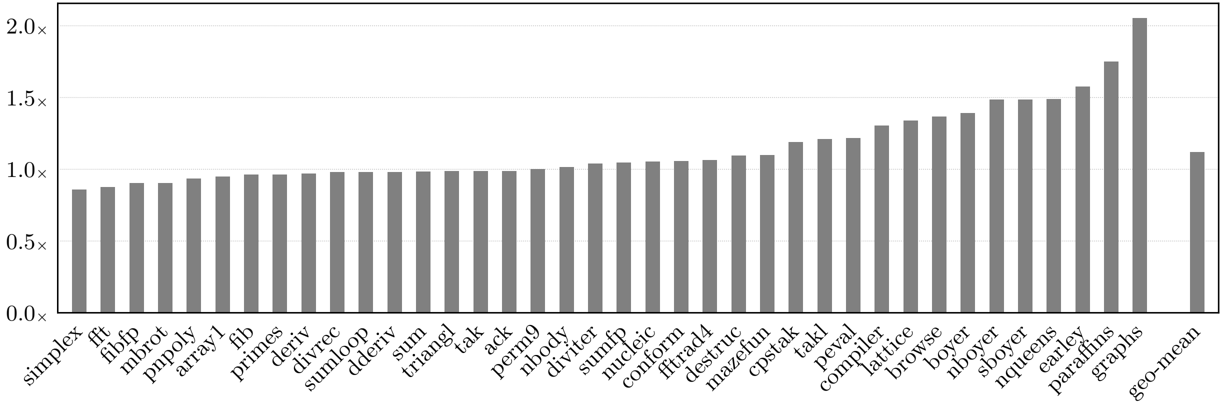


Fig. 6.12. Compilation time of the generated code using interprocedural BBV relative to intraprocedural BBV for LC with a version limit of 5 (intraprocedural BBV is 1.0).

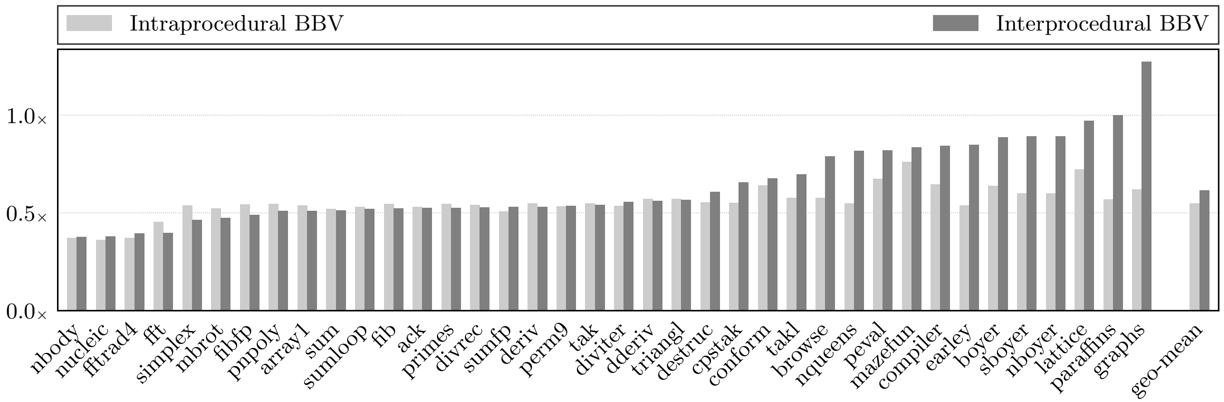


Fig. 6.13. Compilation time of the generated code using intraprocedural and interprocedural BBV relative to naive compilation for LC with a version limit of 5 (naive compilation is 1.0).

`earley` is the benchmark that is the most impacted by this limit. For this benchmark, the execution time using interprocedural code specialization is 0.69× intraprocedural BBV with a limit of 5 and 0.62× with no limit. The limit does not significantly impact the performance of the other benchmarks.

6.3.5. Compilation Time

Figure 6.12 shows the compilation time of the benchmarks executed by LC using interprocedural code specialization relative to the compilation time using intraprocedural code specialization (intraprocedural specialization is 1.0). We observe an expected strong correlation between the compilation time and the size of the generated code. For some

benchmarks, the compilation time is slightly decreased because, for these benchmarks, most of the code is monomorphic. Using interprocedural specialization, the single version that is generated is more optimized, less code is generated making it faster to compile. The biggest decrease is for the `simplex` benchmark. For this benchmark, the compilation time using the interprocedural extensions is 0.86× intraprocedural specialization.

For about half of the benchmarks, the compilation time is increased. The worst cases are the same than the worst cases of the code size graph. These benchmarks use more polymorphic code. Because the extensions allow the compiler to discover more properties, it discovers more types than when using intraprocedural BBV resulting in an increase in the number of generated versions. Because more properties are known, these versions are more optimized and smaller, but because there are more versions, more code is generated and the compilation time is increased. The `graphs` benchmark is the most affected by this increase. The compilation time of this benchmark using the interprocedural extensions is 2.05× intraprocedural specialization.

An important issue with our implementation is that we focused the development effort on optimizing the generated code and not on optimizing the code generation process. With a compiler designed for code duplication, optimizing the code generation process may be more important than with other JIT compiler designs. Exploring the optimization of the code generation process in the context of JIT compilation based on interprocedural code specialization is an issue that would be worth addressing in the future.

The compilation time using interprocedural code specialization is 1.12× intraprocedural code specialization on average.

Figure 6.13 shows the compilation time of the benchmarks executed by LC using interprocedural specialization relative to the compilation time using naive compilation (intraprocedural specialization is 1.0). The graph also shows the compilation time of the benchmarks executed by LC using intraprocedural specialization in light gray for reference.

We see that, similarly to the code size, the increase of the compilation time is mitigated by the significant decrease we observed with intraprocedural specialization. Compared to naive compilation, all the benchmarks except three are faster to compile with the interprocedural extensions. For two of these three benchmarks (`lattice` and `paraffins`), the compilation time is similar to the compilation time of naive compilation. A significant increase in the

compilation time is observed only for the `graphs` benchmark. The compilation time of this benchmark is 1.27× the compilation time of naive compilation.

The `nbody` benchmark is the best case. The compilation time of this benchmark using the interprocedural extensions is 0.38× the compilation time of naive compilation.

Overall, the code is significantly faster to compile using interprocedural BBV than naive compilation. The compilation time is 0.62× naive compilation on average.

6.3.6. Summary

These experiments show that the interprocedural extensions allow generating faster code than both naive compilation and intraprocedural specialization.

We also observed a decrease in the size of the generated code and in the compilation time compared to naive compilation and, because more properties are discovered and more versions are generated, we observed an increase compared to intraprocedural specialization.

These observations indicate that a JIT using interprocedural specialization probably requires a greater warmup time than a JIT based on intraprocedural only specialization, but it allows generating more optimized and faster code. An advantage of our implementation is that, because of its design, this warmup stays lower and more predictable than other JIT compilation approaches. Because the compiler uses a single compilation level, no recompilation and no interpretation nor profiling phase, the warmup time is limited to the compilation time of the versions of the single compilation level.

Our experiments also show that the space used by our table-based implementation of the dynamic dispatch is not significant but must be considered in context in which memory usage is an important concern.

Finally, similarly to intraprocedural BBV, limiting the number of versions to 5 does not significantly impact the quality of the code generated by interprocedural BBV.

6.4. Eager Unboxing

In this section, we evaluate the impact of eager unboxing in LC. As explained in Chapter 4, our implementation uses Tagging and eagerly unboxes flonums using BBV. To evaluate the technique, we analyze the number of executed boxing and unboxing operations and we measure its impact on the execution and compilation time.

	LC - Naive unboxing		LC - Eager unboxing			
Benchmark	# boxing	# unboxing	# boxing	# unboxing	% boxing	% unboxing
fibfp	89582129	179164234	9	14	≈0	≈0
mbrot	137762923	273654718	7	11	≈0	≈0
sumfp	400040023	800100020	9	20012	≈0	≈0
fftrad4	199566697	390919526	41764183	60116308	21	15
nbody	300000381	590000698	75000012	170000273	25	29
fft	54382023	108638020	18432007	36864012	34	34
nucleic	34633209	68644606	6151038	31196272	18	45
pnpoly	18000023	194200018	7	94100011	≈0	48
simplex	20800023	42400018	10800007	28900011	52	68

Tab. 6.2. Number of executed flonum boxing and unboxing operations, using heterogeneous vectors, with eager unboxing relative to naive mode.

Some benchmarks intensively using flonums also use vectors to store intermediate computations. Although they are not part of the Scheme standard, most Scheme implementations offer *f64vectors* (homogeneous vectors of double precision flonums). Consequently, we decided to measure the impact of the technique in two configurations. In the first configuration, the benchmarks use f64vectors where possible and in the second configuration, the benchmarks use the normal heterogeneous vectors.

6.4.1. Number of Executed Boxing and Unboxing Operations

6.4.1.1. *Heterogeneous Vectors*

Table 6.2 shows the number of executed flonum boxing and unboxing operations for each benchmark using heterogeneous vectors. Note that the results are presented for the flonum intensive benchmarks only because no flonum operations are executed for the others. The first two columns show the results for LC using naive unboxing (i.e. the naive approach in which the primitive's arguments are always unboxed before the primitive operation and the result always boxed immediately afterward). The next two columns show the results for LC using eager unboxing. The last two columns in light gray, show the results for LC using eager unboxing relative to LC using naive unboxing. For the results presented in this table, we used a limit of 5 versions. We haven't observed any significant difference in the results with a higher limit (including with no limit).

	LC - Naive unboxing		LC - Eager unboxing			
Benchmark	# boxing	# unboxing	# boxing	# unboxing	% boxing	% unboxing
fft	93168023	128990020	8	12	≈0	≈0
fftrad4	262133765	435134470	8	12	≈0	≈0
fibfp	89582129	179164234	9	14	≈0	≈0
mbrot	137762923	273654718	7	11	≈0	≈0
nbody	470000641	665000701	9	13	≈0	≈0
pnpoly	112100023	194200018	7	11	≈0	≈0
sumfp	400040023	800100020	9	20012	≈0	≈0
nucleic	65971759	74996176	251538	209332	<1	<1
simplex	48300023	52800018	1400007	2400011	3	5

Tab. 6.3. Number of executed flonum boxing and unboxing operations, using f64vectors, with eager unboxing relative to naive mode.

With naive compilation, no optimization is performed to remove boxing and unboxing operations. This configuration then represents the worst case of a naive code generation from an AST to the target language.

We see in the table that there are two groups of benchmarks. For the first group (`fibfp`, `mbrot`, `sumfp`), almost all the boxing and unboxing operations are removed using eager unboxing. For the other group (`fftrad4`, `nbody`, `fft`, `nucleic`, `pnpoly`, `simplex`), a significant number of operations are removed (between 32% and 85%).

The benchmarks of the second group use vectors to store intermediate results. Because the compiler does not track the type of the values in vectors, each time a flonum is written to a vector, a boxing operation is executed. Moreover, each time a flonum is fetched from a vector, its type is unknown and the compiler has to discover it again. Consequently, eager unboxing removes fewer boxing and unboxing operations for these benchmarks.

Eager unboxing removes 83% of the boxing operations and 73% of the unboxing operations on average.

6.4.1.2. *Homogeneous Vectors*

Table 6.3 shows the number of executed flonum boxing and unboxing operations for each benchmark using homogeneous vectors. The first two columns show the results for LC using naive unboxing. The next two columns show the results for LC using eager unboxing. The last two columns in light gray show the results for LC using eager unboxing relative to LC

using naive unboxing. For the results presented in this table, we also limited the number of versions to 5. We haven't observed any significant difference in the results with a higher limit either (including with no limit).

An interesting effect can be observed in this configuration. For all the benchmarks, almost all boxing and unboxing operations are removed. Because the compiler discovers and interprocedurally propagates the types, when a flonum with known type is written in a homogeneous vector, it is already handled unboxed and no unboxing operation is necessary to write it to the vector. When a flonum is fetched from a flonum homogeneous vector, the compiler knows that it is a flonum. Its type is then propagated and this flonum is never unboxed while its type is known.

This observation is interesting because homogeneous vectors are generally used for their convenience for interfacing with low-level libraries [**53**] and generally negatively impact the performance of the generated code. Indeed, the compilers typically generate more boxing and unboxing operations to store values in homogeneous vectors in their unboxed representation. However, in the case of flonums, if the boxed flonums are memory allocated, using homogeneous flonum vectors improves GC time because they are stored unboxed.

Using homogeneous flonum vectors and eager unboxing, the generated code is more optimized because the compiler uses the type of the homogeneous vector to remove almost all of the boxing and unboxing operations. In addition, the positive impact of using homogeneous vectors on the GC time is preserved.

The overall performance is significantly improved when using eager unboxing, in particular with the use of homogeneous vectors. These experiments show that the use of homogeneous vectors allows our implementation to generate code for flonum benchmarks that is as much optimized as when using a value representation with no impact, such as NaN-boxing, because in both cases, almost no operation is executed. However, unlike NaN-boxing, using eager unboxing does not impact the performance of values of other types.

To compare the impact of eager unboxing with local CSE, we compare the number of executed operations with the Gambit Scheme compiler. For this experiment, we used Gambit v4.9.3. Because LC uses Gambit's frontend, the input AST is the same for LC and Gambit (Gambit's default inlining configuration is used). Note that Gambit is configured in safe mode for this experiment. Using safe mode makes the basic blocks smaller and Gambit's

	LC - Naive unboxing		Gambit			
Benchmark	# boxing	# unboxing	# boxing	# unboxing	% boxing	% unboxing
nucleic	34633209	68644606	26485033	60147201	76	88
nbody	300000381	590000698	280000458	530001008	93	90
mbrot	137762923	273654718	137763039	239584509	≈100	88
fft	54382023	108638020	54382139	108620406	≈100	≈100
fftrad4	199566697	390919526	199566819	393017083	≈100	≈100
fibfp	89582129	179164234	89582243	179164616	≈100	≈100
pnpoly	18000023	194200018	18000136	194200398	≈100	≈100
simplex	20800023	42400018	20800139	42400404	≈100	≈100
sumfp	400040023	800100020	400040140	800100413	≈100	≈100

Tab. 6.4. Number of executed flonum boxing and unboxing operations, using heterogeneous vectors, with the Gambit Scheme compiler relative to LC in naive mode.

local CSE less effective, but it allows us showing how LC compares to other implementations with similar configurations. For a better comparison with LC, we configured Gambit to use block compilation (*block* declaration) and standard Scheme bindings (*standard-bindings* declaration).

LC using naive unboxing represents the worst case because in this configuration no optimization is used to remove a single boxing and unboxing operation. Gambit uses local CSE within basic blocks.

Table 6.4 shows the number of executed flonum boxing and unboxing operations for each benchmark using heterogeneous vectors. The first two columns show the results for LC using naive unboxing. The next two columns show the results for Gambit. The last two columns in light gray, show the results for Gambit relative to LC using naive unboxing. The results are slightly better for Gambit with heterogeneous vectors than with f64vectors but the difference is not significant.

We see in the table that the local CSE of Gambit allows removing up to 23.5% of the boxing operations and 12.4% of the unboxing operations. For the other benchmarks, local CSE has no significant impact. Even if a local CSE allows significantly removing operations, its impact is much less significative than a completely dynamic approach relying on eager unboxing of values based on BBV.

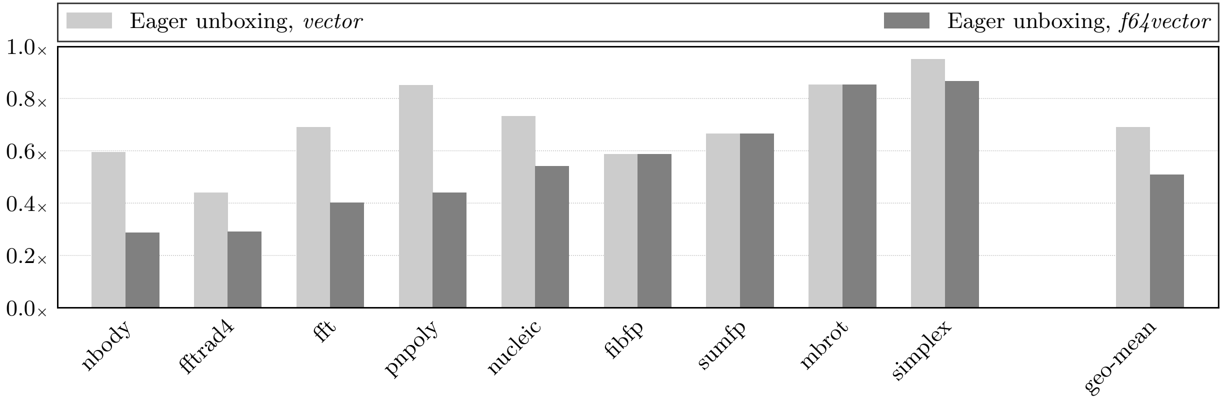


Fig. 6.14. Execution time of the generated code of flonum benchmarks using eager unboxing (with normal heterogeneous vectors and f64vectors) relative to naive unboxing (respectively with heterogeneous vectors and f64vectors) for LC with a version limit of 5 (naive unboxing is 1.0).

6.4.2. Execution Time

Figure 6.14 shows the impact of eager unboxing on the execution time of the generated code. The figure shows the execution time of flonum intensive benchmarks executed with LC using eager unboxing relative to LC using naive unboxing (naive unboxing is 1.0). The light gray bars show the results using heterogeneous vectors. The dark gray bars show the results using homogeneous vectors.

We see that in both cases, using eager unboxing significantly improves the performance of all the benchmarks. For the reasons mentioned in the previous section, eager unboxing is more significantly impacting the benchmarks executed with homogeneous vectors. `nbody` is the best case. The execution time for this benchmark using eager unboxing is 0.29× naive unboxing. With heterogeneous vectors, `fftrad4` is the best case. The execution time for this benchmark using eager unboxing is 0.44× naive unboxing. With both heterogeneous and homogeneous vectors, the worst case is `simplex` with a factor of 0.95× with heterogeneous vectors and 0.87× with homogeneous vectors. Performance is significantly improved by a factor of 0.69× with heterogeneous vectors and 0.51× with homogeneous vectors on average.

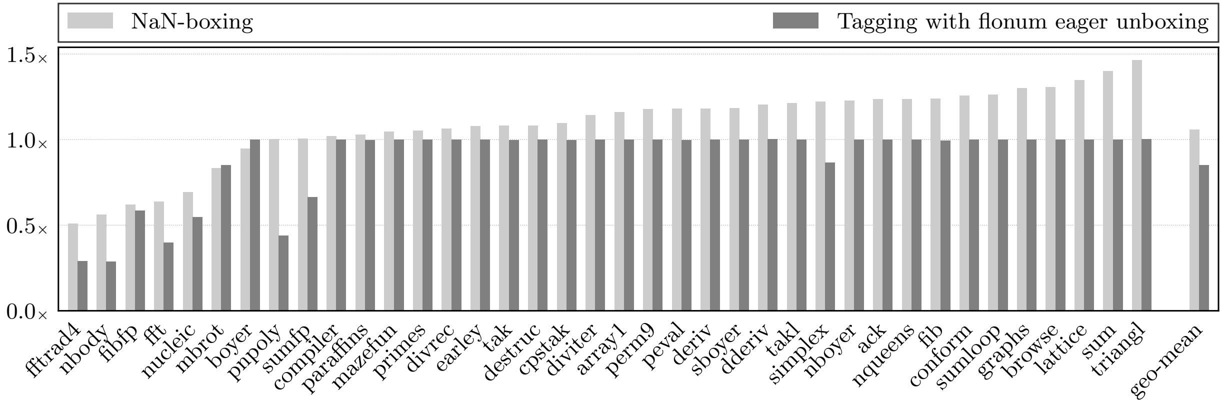


Fig. 6.15. Execution time of the generated code using NaN-boxing and Tagging with eager unboxing relative to Tagging without eager unboxing for LC with a version limit of 5 (Tagging without eager unboxing is 1.0).

Figure 6.15 shows the execution times of the code generated with NaN-boxing and with Tagging used with flonum eager unboxing relative to the execution time of the code generated by LC using Tagging without eager unboxing (Tagging without eager unboxing is 1.0). Note that the execution times presented in this figure may slightly differ from the times presented in the other sections. To make a fair comparison between Tagging and NaN-boxing, we have disabled an optimization LC does on the Tagging representation for these executions. For these results, the flonum benchmarks are executed with homogeneous vectors.

The light gray bars show the execution time of the code generated by LC using NaN-boxing. NaN-boxing allows generating code that is much more efficient for the benchmarks using flonums. However, the other benchmarks are slightly less efficient using NaN-boxing than naive Tagging. Overall, the execution time of the code generated with NaN-boxing is slightly increased compared to the execution time of the code generated with Tagging. The execution time with NaN-boxing is 1.06× the execution time of pure Tagging on average.

The dark gray bars show the execution time of the code generated by LC using Tagging with eager unboxing of flonums. This approach allows generating code that is much more efficient than the code generated by pure Tagging for the benchmarks using flonums. We explained that using eager unboxing and Tagging, the performance of flonum operations is improved without impacting the performance of the operations on values of other types. Because some flonum benchmarks also intensively use values of other types, they are faster

than when using NaN-boxing. For example, the `fftrad4` benchmarks intensively uses fixnum and vector operations in addition to flonum operations and is thus significantly faster than when NaN-boxing is used. Moreover, this figure shows that eager unboxing of flonums does not impact the performance of non-flonum benchmarks thus allows generating significant more efficient code compared to NaN-boxing.

Overall, Tagging with flonum eager unboxing allows generating code that is significantly more efficient than both pure Tagging and NaN-boxing. The execution time with Tagging and eager unboxing is 0.85× pure Tagging and 0.81× NaN-boxing on average.

These results validate the approach we presented in Chapter 4. They show that in a context in which development effort is an important concern, relying on the optimization opportunities of Tagging allows focusing the development effort on optimizing flonums only to generate code that is significantly faster than when naively using existing representations for both flonum and non-flonum benchmarks.

6.4.3. Compilation and Garbage Collection Time

Using Tagging, GC time for flonum intensive benchmarks may be high because flonums are memory allocated. NaN-boxing completely eliminates the GC time with both homogeneous and heterogeneous vectors. With Tagging and eager unboxing, most of the flonums are handled unboxed and they are only memory allocated when the compiler is not able to track their type, which occurs rarely with these benchmarks. Using Tagging with eager unboxing, the GC time is also eliminated if the benchmarks are executed with homogeneous vectors. Using heterogeneous vectors, more flonums are boxed and the GC time is almost but not completely eliminated. In this case, the GC time using eager unboxing is only 0.08× the GC time of pure Tagging.

Using flonum eager unboxing does not significantly impact the compilation time. On average, the compilation time for naive Tagging, NaN-boxing and Tagging with eager unboxing is almost the same using both homogeneous and heterogeneous vectors.

6.4.4. Summary

Interprocedural eager unboxing allows removing a significant number of boxing and unboxing operations. We showed that, when applied to flonums, it allows removing almost all of the operations for some benchmarks and almost all of the operations for all the flonum

benchmarks if homogeneous vectors are used. Consequently, in an implementation based on Tagging such as LC, eager unboxing of flonums allows generating code that is significantly more efficient for flonum operations, without impacting the performance of the operations on values of other types.

Using eager unboxing only on flonums offers a good trade-off between the overall performance of the system and development effort. Using this approach, the overall performance is significantly improved compared to naive implementation of Tagging and NaN-boxing by focusing the effort on a single type.

In a context in which development effort need not be kept low, it would be interesting to analyze the impact of eager unboxing based on BBV on the values of other types. This issue is presented in Chapter 7.

6.5. General Performance

In this section, we compare the performance of LC against mature implementations of the Scheme language, namely Gambit, Chez Scheme and Pycket [**2**]. This comparison gives an idea of the overall performance a relatively simple compiler, that did not require a significant development effort, can achieve.

6.5.1. Ahead-Of-Time Optimizing Compilation

Here, we evaluate the performance of the generated code. We compare the performance of the code generated by LC with the code generated by the Gambit Scheme compiler v4.9.3 (February 2019) and the Chez Scheme compiler v9.5.2 (March 2019). Gambit and Chez are two mature AOT Scheme implementations that are known to be among the most efficient implementations of the Scheme language.

It is worth mentioning that unlike these implementations, LC is a research-oriented implementation. This means that, even it they are not part of the Scheme standard, several production-level features that are classically offered by dynamic language are implemented in Gambit and Chez but not in LC. This is typically the case for multi-tasking and stack-overflow detection. Previous study showed that these features can be implemented with polling, causing an overhead of 15% on average for Gambit [**52**]. Although this study analyzed

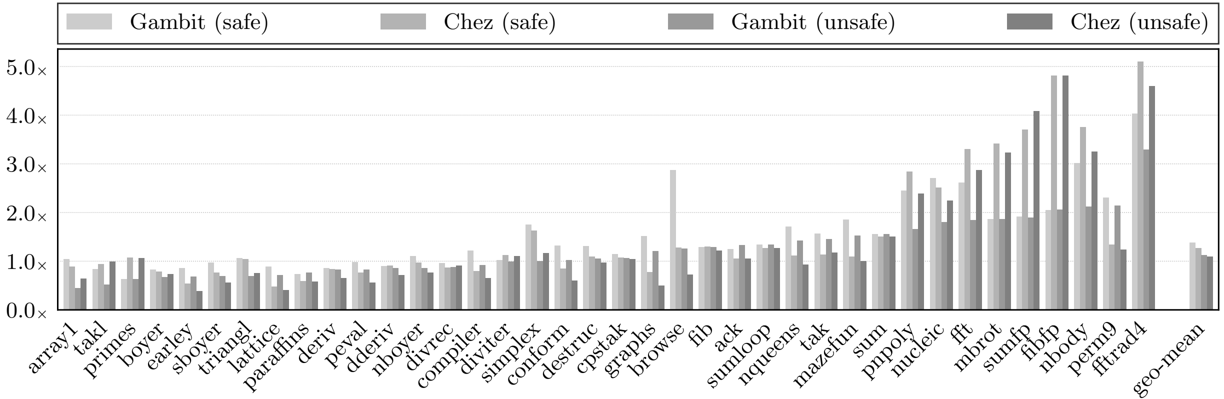


Fig. 6.16. Execution time (excluding GC and compilation time) of the code generated by Gambit and Chez in safe and unsafe modes relative to LC (LC is 1.0).

the overhead for an old architecture, we can imagine that, by not implementing these features, LC gets a small performance boost compared to these implementations.

For this experiment, LC uses interprocedural code specialization, Tagging and eager unboxing of flonums. The optimizations presented in Section 2.10 are applied intraprocedurally, only the types are propagated interprocedurally. The number of basic block versions is limited to 5.

For a better comparison, Gambit uses block compilation (*block* declaration) and standard Scheme bindings (*standard-bindings* declaration). For its unsafe mode, we used the (*not safe* declaration).

For this experiment, we used a single-threaded Chez Scheme build. We used the option *optimize-level 2* to compile the benchmarks in safe mode and the option *optimize-level 3* for the unsafe mode. To use the R5RS Scheme standard, we used the following import:

(import (rnrs) (rnrs mutable-pairs) (rnrs mutable-strings) (rnrs r5rs))

Finally, we used the `--program` option of Chez Scheme to ensure each benchmark is executed as an RNRS top-level program.

Figure 6.16 shows the execution time of the code generated by Gambit and Chez in safe and unsafe modes relative to LC (LC is 1.0). We see that the code generated by LC is faster for more than half of the benchmarks.

The best case for LC is `fftrad4`. For this benchmark, the execution time using Gambit in safe mode is 4.03× LC and 3.29× in unsafe mode. The execution time of this benchmark using

Chez in safe mode is 5.10× LC and 4.60× in unsafe mode. We see that the code generated by LC is significantly faster than Gambit and Chez in safe mode, the execution time with Gambit is 1.38× LC and the code generated by Chez is 1.27× LC on average. However, we mentioned that LC does not perform as many checks as these implementations when they are in safe mode. But we see that LC is also faster than Gambit and Chez in unsafe mode. In unsafe mode, the execution time with Gambit is 1.12× LC and the code generated by Chez is 1.09× LC on average.

For this experiment, the flonum benchmarks use heterogeneous vectors. We also compared the execution time of the code generated by Gambit and LC using f64vectors. In this situation, the results are similar, but slightly more beneficial for LC.

This experiment suggests that our techniques allow generating code that competes with the code generated by mature implementations of Scheme. The results also highlight the necessity of implementing strongly dynamic languages with JIT and dynamic techniques. Scheme is a dynamic language that is traditionally implemented by AOT compilers. However, these results show that using simple purely dynamic techniques allows generating code that, in several cases, is more efficient than more complex and mature implementations.

These observations reinforce the idea that using even more dynamic techniques, in particular extending the use of interprocedural specialization to other properties, would allow further improving the performance of these languages.

6.5.2. Just-In-Time Optimizing Compilation

In this section, we evaluate the general performance of the system. We compare the overall performance of LC with the performance of Pycket (February 2019 build). Pycket is a JIT implementation of Racket [**55**], a dialect of Scheme and is built with the RPython framework [**109**]. It is the only relatively mature Scheme implementation offering an optimizing JIT compiler (tracing JIT) and it is actively maintained. The flonum benchmarks are executed with heterogeneous vectors. We compare the total execution time of the benchmarks using LC with the total execution time of the benchmarks using Pycket. The total execution time includes execution and JIT compilation time but excludes GC time. For this comparison, LC uses the same configuration as the one used for the previous experiment.

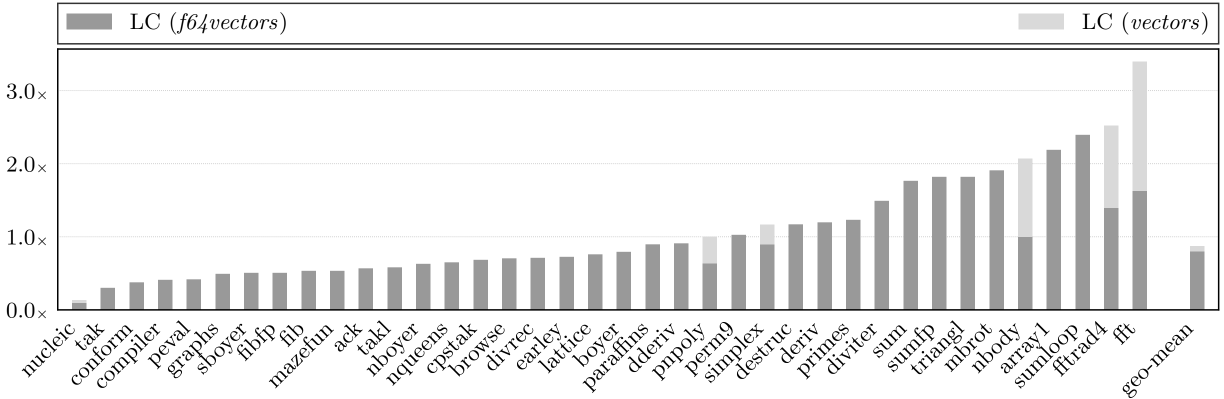


Fig. 6.17. Execution time (excluding GC time and including JIT compilation time) of the code generated by LC relative to Pycket (Pycket is 1.0). The dark gray bars show the execution time for LC using f64vectors and the light gray bars show the difference using heterogeneous vectors.

Figure 6.17 shows the execution time (excluding GC time and including JIT compilation time) of the code generated by LC, using f64vectors and heterogeneous vectors, relative to Pycket. We see that the benchmarks are faster with LC for 22 out of 38 benchmarks. With eager unboxing of flonums, some flonum benchmarks (`nucleic`, `fibfp` and `pnpoly`) have the same or better performance with LC. However, the other flonum benchmarks are faster with Pycket because tracing optimizes loops particularly well and these benchmarks intensively use explicit loops. LC does not put particular effort into detecting and optimizing loops. With LC, the execution time of `compiler` is 0.41× the execution time of Pycket. The best case for LC is `nucleic`. Its execution time using LC is 0.13× the execution time of Pycket (0.09× with f64vectors). The worst case is `fft`. For this benchmark, the factor is 3.40× (1.63× with f64vectors). We see that using f64vectors, the flonum intensive benchmarks are significantly faster. The execution time of the code generated by LC is 0.87× the execution time of the code generated by Pycket on average (0.80× with f64vectors).

Using simple techniques allows LC to execute Scheme programs with good performance competing with other JIT Scheme implementations, even though no effort has been put on optimizing the compilation time. This experiment suggests that our simple approach, giving full control over the compilation process without strong dependence to other systems and with relatively low development effort offers very good performance.

6.6. Summary

Our experiments first show that BBV allows generating efficient code for the Scheme programming language. They also show that intraprocedural BBV, as implemented with our design, allows generating code that is significantly more efficient than naive compilation from AST, with relatively low development effort.

Using our interprocedural extensions, the compiler is able to discover much more dynamic properties of the executed programs allowing generating more efficient code than intraprocedural BBV.

We also showed that by relying on the optimization opportunities of the value representations, we can focus on optimizing flonums only to generate code that is overall significantly more efficient than when naively using these representations, again with relatively low development effort.

Finally, our experiments suggest that limiting the development effort and the system complexity when implementing a programming language does not necessarily imply poor performance. We showed that our implementation is often more efficient than existing mature JIT and AOT optimizing implementations of Scheme.

These observations support the thesis that using these dynamic language compilation techniques is a good choice in a context in which development resources are limited.

Chapter 7

Future Work

During our research, we identified several ways to extend our work. In this chapter, we outline the opportunities that are most promising.

7.1. Compound Data Types

In our implementation, BBV does not track the type of values in compound data types. The interprocedural extensions we have developed allow discovering significantly more types than intraprocedural BBV but, even when using these extensions, our implementation of BBV has difficulty optimizing the benchmarks intensively using compound data types. It would be interesting to explore the propagation of these types, in particular for Scheme vectors and pairs.

For vectors, the compiler may specialize the code using the type of the values in the vectors. In addition to the types, the compiler could specialize the code using the length property to optimize vector accesses. A similar approach can be used for pairs. However, using this approach could cause an explosion in the number of versions for functions using nested pairs and vectors with values of several different types.

It would be interesting to explore new approaches to limit this explosion. For example, the compiler may limit the specialization to homogeneous lists and vectors or limit the number of nested pairs for which it tracks the type of the elements.

```
1:  (define (proper-divisors-h n m)
2:    (cond ((= m 1)
3:            '())
4:          ((= (remainder n m) 0)
5:            (cons m (proper-divisors-h n (- m 1))))
6:          (else
7:            (proper-divisors-h n (- m 1)))))
8:
9:  (define (proper-divisors n)
10:   (proper-divisors-h n (- n 1)))
11:
12: (define (sum d)
13:   (if (null? d)
14:       0
15:       (+ (car d) (sum (cdr d)))))
16:
17: (let ((divisors (proper-divisors 100)))
18:   (let ((s (sum divisors)))
19:     (display divisors)
20:     (newline)
21:     (display s)
22:     (newline)))
```

Fig. 7.1. Scheme code to compute the proper divisors of a fixnum and the sum of these divisors.

Figure 7.1 shows code for which limiting the specialization to homogeneous lists allows generating efficient code. The `proper-divisors` function computes, using the `proper-divisors-h` helper function, the list of the proper divisors of its argument `n`. For each recursion step of the `proper-divisors-h` function, the divisor `m` is added to the list if it is a proper divisor. We assume that when specializing the code for homogeneous lists, the compiler propagates types based on the following typing rules:

```
1. (cons T null)          => (list-of T)
2. (cons T (list-of T))   => (list-of T)
3. (car (list-of T))      => T
4. (cdr (list-of T))      => (list-of T)
```

Using interprocedural BBV and these typing rules, there are four possible cases when generating the code of this function:

- The compiler generates the base case, the function returns the empty list, the compiler propagates this type to the continuation.
- `m` is a proper divisor, it is added to the list of divisors computed by the recursive call. This list contains no divisor meaning that its propagated type is *null*. The compiler also knows that `m` is a fixnum. It applies Rule 1 and propagates the type *(list-of fix)* to the continuation.
- `m` is a proper divisor, it is added to the list of divisors computed by the recursive call. This list contains at least one divisor meaning that its propagated type is *(list-of fix)*. The compiler knows that `m` is a fixnum, it applies Rule 2 and propagates the type *(list-of fix)* to the continuation.
- `m` is not a proper divisor, the type of the value returned by the recursive call is propagated to the continuation (either *null* or *(list-of fix)*).

With interprocedural BBV, the type tests are eliminated for the `proper-divisors-h` function and the compiler knows the type of the value it returns thus it knows the type of the `divisors` variable (either *null* or *(list-of fix)*).

The `sum` function computes the sum of the proper divisors of `n`. Assuming that there is at least one divisor in the list, the call to this function causes the compiler to generate a version specialized for `d` being a *(list-of fix)*. The compiler generates the code to test if the argument is null. If the test is false, the compiler knows that `d` is a *(list-of fix)* containing at least one fixnum meaning that no type test is needed for the `car` and `cdr` primitives, it knows that `d` is a pair. No type test is needed for the operands of the addition because the compiler knows that `(car d)` is a fixnum (Rule 3) and it knows the type of the value returned by the recursive call. Moreover, the compiler knows that the result of `(cdr d)` is a *(list-of fix)* with Rule 4 (of either 0 or more fixnums). Consequently, the same version of the function is used for the recursive call. Using this approach, a single type test for null, is generated for the `sum` function.

An issue exists if the compiler propagates the type of the values in a mutable compound data type that escapes. In this situation, the compiler cannot guarantee the type after the data type escaped. This issue can be addressed by Typed Object Shapes [**143, 38**]. Type Object Shapes solve this issue but require additional shape tests. An escape analysis can also be performed to determine that the type of the list can safely be propagated. Another

solution, particularly adapted to functional languages, would be to apply this optimization to immutable values only. For example, to use this solution for pairs, it is possible to distinguish operations on mutable pairs from operations on immutable pairs as is the case in Racket in which pairs are also immutable by default.

7.2. Properties Used for Specialization

As presented in Chapter 5, our implementation uses BBV to intraprocedurally propagate constants, types and register allocation information, and types are also interprocedurally propagated.

Our experiments showed that propagating these properties does not cause a noticeable explosion in the number of generated versions and allows generating more efficient code. However, it would be interesting to explore the impact of propagating more properties.

7.2.1. Constants

In our work, we explored the interprocedural propagation of constants. Propagating constants allows generating code that is much more efficient for some cases but results in an explosion in the number of versions for most of the benchmarks. The main reason is that our implementation is not able to detect loops that are usually expressed with recursive functions in Scheme. When a number is used as a loop counter and a compiler propagating constants knows its initial value, the loop is specialized for each value the counter takes (i.e. it is specialized `n` times, `n` being the number of iterations of the loop). To avoid this explosion, a simple compiler limiting the number of versions falls back to a completely generic version. Because of this overspecialization, this version is used after some iterations for the rest of the iterations, degrading performance. Note that if `n` is less than the version limit this would instead be optimal.

Using a generic fallback version when the limit is reached is not optimal in this situation. This approach with abrupt degradation is illustrated on the top part of Figure 7.2.

To avoid this issue, several solutions can be used.

- Static analyses can be used to detect loops to avoid overspecializing them. However, loop constructs can be hard to detect for Scheme if they are expressed by nested recursive functions, and low iteration count loops are interesting to optimize.

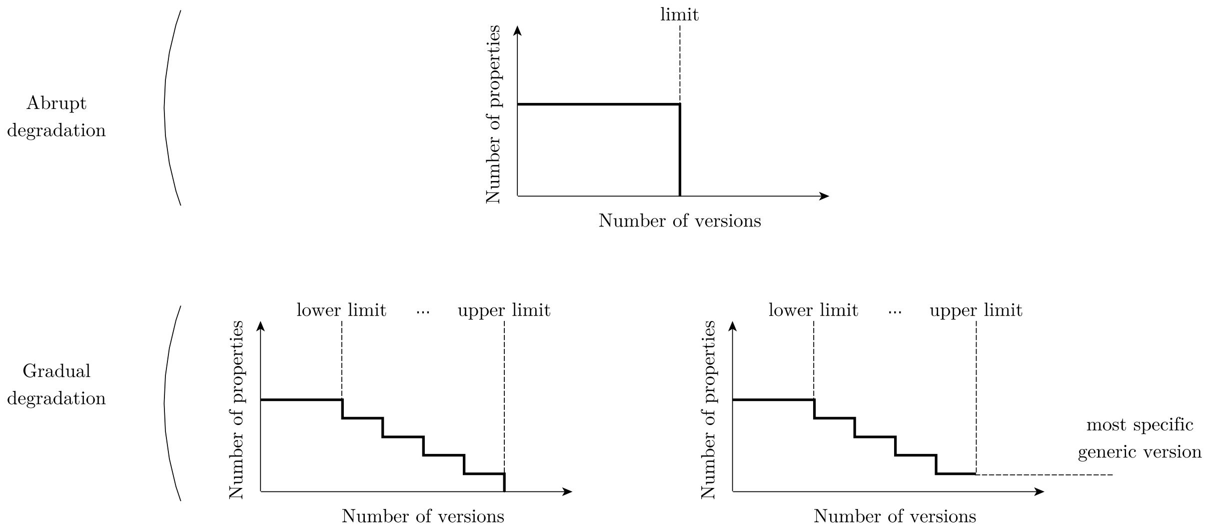


Fig. 7.2. Three approaches to implement versioning while avoiding explosions in the number of versions. The approach on the top part using abrupt degradation illustrates how a hard limit avoids the explosion. The approach on the left side illustrates how heuristics allow gradually reaching the generic version. The approach on the right side illustrates how using a generic version bounded by a static BBV phase avoids falling back to a completely generic version.

- The compiler could use heuristics to avoid falling back to a completely generic version. For example, it could limit the number of versions specialized for constants and fallback to a version specialized for types and register allocation only when it is reached. The compiler can use another limit for these versions. Using the same process, the compiler can fallback to more generic versions until the completely generic version is reached. The left side of Figure 7.2 illustrates this gradual degradation approach. We see that the generic version is progressively reached. We explored this solution to avoid the explosion for loops but we observed that it increases the compilation time.
- In addition to the heuristics, the compiler could use an abstract interpretation of the program before its execution using code duplication similar to BBV to track properties more precisely to compute, for each block, its most generic fallback context. This would allow reaching a generic version faster and avoiding the increase in the compilation time without falling back to a completely generic version. In the case of loops, this phase could detect that it is useless to specialize the code for a loop

counter but detect that this counter is of a fixed type. The compiler can then use this observation when generating the generic fallback version. The right side of Figure 7.2 illustrates this approach.

7.2.2. Register Allocation

Another research direction is the interprocedural propagation of the register allocation property. We explained that our implementation is able to pass unboxed flonum arguments in the floating point registers to the callee functions when generating call sites. This is an implicit form of interprocedural specialization based on the register allocation property, this property is lazily inferred from the propagated types. However, as in other systems, our implementation follows a specific calling convention. To generate a function call, the arguments are moved to their associated argument locations (register or stack). To avoid these moves, the compiler could directly pass the arguments in the registers where they currently are when generating the call site and propagate this register allocation property to the callee function that can be specialized for this property.

However, this optimization adds new issues. In particular, if an argument is modified in the callee function, this modification must not be effective in the call continuation. This issue is typically solved by generating additional instructions to save the live arguments to the execution stack before generating call sites but these additional instructions impact performance.

A solution relying on BBV would be to delay this save until the callee function actually modifies the value of an argument. When the callee function modifies it, the compiler could generate code to save the previous value to another location and propagate this new location to the call continuation. In addition to removing the unnecessary save operations, this lazy solution allows using the available registers instead of the stack to save the arguments, possibly improving the performance of the generated code.

7.3. Lazy Unboxing in Other Contexts

Our exploration of unboxing based on BBV focused on generating efficient code with low development effort. Our experiments showed that wisely choosing a value representation

and using eager unboxing based on the properties discovered and propagated by BBV allows generating code that is more efficient than when naively using existing representations.

It would be interesting to extend this exploration to a context in which development effort is not an important concern (i.e. for a compiler intensively using static analyses and a multi-tier architecture in addition to dynamic techniques) by applying it to values of other types. Eager unboxing can for example be applied to fixnums and memory allocated objects in addition to flonums. We explained that even if Tagging is more efficient at handling fixnums than NaN-boxing, some operations such as multiplication still require one of the arguments to be unboxed. When eager unboxing is applied to fixnums, this unboxing operation can be removed. Using eager unboxing on fixnums and memory allocated objects would also allow significantly improving performance when NaN-boxing is used.

This global exploration of unboxing based on BBV would allow to determine, for dynamic languages, how important the choice of a run time value representation is for an optimizing JIT compiler that knows the types in most cases and for which the development effort is not an important concern.

Chapter 8

Conclusion

Building optimizing JIT compilers for programming languages typically requires a significant development effort, in particular for dynamic languages. Our thesis is that VMs for higher-order dynamic languages can be based on BBV, with relevant extensions, to achieve competitive performance with relatively low development effort.

We developed a compilation design using CPS to represent the compilation process (Chapter 2). This design allows generating code directly from ASTs without transforming the source program to other intermediate representations and allows implementing BBV. We explained how this design can easily be extended to apply some classical optimizations. This approach allows implementing optimizing JIT compilers for programming languages in a similar way to naive tree walking interpreters and code generators, significantly decreasing the development effort.

We also extended BBV to interprocedurally specialize the generated code in the presence of higher-order functions (Chapter 3). This technique extends classical closure and continuation representations to store multiple specialized entry points. This extension propagates the properties discovered by BBV through function calls and returns, offering several optimization opportunities.

We explained how compilers can rely on existing run time representations of values to optimize values of specific types allowing to focus the development effort on values of a small set of types (Chapter 4). Using Tagging and interprocedural eager unboxing of flonums, the negative impact on performance of boxing and unboxing operations can significantly be decreased. This allows generating code that is more efficient than when naively using these representations, without significantly impacting development effort.

We explained how we reused some components of an existing Scheme implementation in LC, our implementation of the Scheme language, to further decrease development effort, without impacting performance (Chapter 5).

We have run experiments to evaluate the techniques and analyze their impact on the generated code (Chapter 6). These experiments allowed us to validate our approach and our techniques. They first suggest that BBV is efficient at implementing the Scheme programming language. They also suggest that the interprocedural extension allows generating code that is significantly more efficient compared to intraprocedural BBV. When applied to typing, it allows removing more than 70% of type tests on average and generating code that is, on average, more than 2× faster than naive code generation from ASTs. The experiments show that flonum eager unboxing removes almost all the flonum boxing and unboxing operations allowing to significantly decrease the overhead of flonum boxing with Tagging. This allows to benefit from Tagging to optimize values of other types and almost eliminates its negative impact on flonums. Using these techniques, LC generates code that, on average, is faster than existing AOT optimizing compilers (Gambit and Chez Scheme) and executes the benchmarks faster than existing JIT implementations (Pycket).

We think these techniques and the results validate our thesis. As implemented with our design, BBV and the interprocedural extensions allowed us to implement a JIT compiler for a dynamic higher-order language, using a single compilation level with a strong focus on decreasing development effort. This implementation achieves relatively good performance that competes with other implementations but is built with relatively low development effort.

That makes this approach particularly attractive in contexts in which relatively good performance is required but development resources are limited.

Bibliography

[1] ChakraCore Javascript Engine. `https://github.com/Microsoft/ChakraCore`.

[2] Pycket Scheme Compiler, February 2019.

[3] SpiderMonkey JavaScript Engine. `https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey`.

[4] The WebKit Open Source Project. `https://webkit.org/`.

[5] V8 JavaScript Engine. `https://v8.dev/`.

[6] IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008*, 2008.

[7] AMD. AMD64 Architecture Programmer's Manual. 2017.

[8] Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 2004.

[9] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 2006.

[10] Andrew W Appel and Trevor Jim. *Optimizing Closure Environment Representations*. Princeton University, Department of Computer Science, 1988.

[11] Andrew W. Appel and Trevor Jim. Continuation-Passing, Closure-Passing Style. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, POPL*, 1989.

[12] Matthew Arnold, Stephen J. Fink, David Grove, Michael Hind, and Peter F. Sweeney. A Survey of Adaptive Optimization in Virtual Machines. *Proceedings of the IEEE*, 93(2), 2005.

[13] J. Michael Ashley and R. Kent Dybvig. An Efficient Implementation of Multiple Return Values in Scheme. In *LISP and Functional Programming*, 1994.

[14] J. Michael Ashley and R. Kent Dybvig. A Practical and Flexible Flow Analysis for Higher-Order Languages. *ACM Transactions on Programming Languages and Systems, TOPLAS*, 20(4), 1998.

[15] John Aycock. A brief history of just-in-time. *ACM Computing Surveys, CSUR*, 35(2), 2003.

[16] Ruben Borisovich Ayrapetyan, Evgeny Aleksandrovich Gavrin, and Andrey Nikolayevich Shitov. The Hybrid Compiler Targeting JavaScript Language. 2017.

[17] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: A Transparent Dynamic Optimization System. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*, 2000.

[18] Spenser Bauman, Carl Friedrich Bolz, Robert Hirschfeld, Vasily Kirilichev, Tobias Pape, Jeremy G. Siek, and Sam Tobin-Hochstadt. Pycket: A Tracing JIT for a Functional Language. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP*, 2015.

[19] Spenser Bauman, Carl Friedrich Bolz, Jeremy Siek, and Sam Tobin-Hochstadt. Sound Gradual Typing: Only Mostly Dead. *Proceedings of the ACM on Programming Languages*, 2017.

[20] Michael Bebenita, Florian Brandner, Manuel Fähndrich, Francesco Logozzo, Wolfram Schulte, Nikolai Tillmann, and Herman Venter. SPUR: a trace-based JIT compiler for CIL. In *Proceedings of the 2010 ACM Conference on Object-Oriented Programming, Systems, Languages and Applications, OOPSLA*, 2010.

[21] Hans-Juergen Boehm and Mark Weiser. Garbage Collection in an Uncooperative Environment. *Software, Practice and Experience*, 1988.

[22] Carl Friedrich Bolz. *Meta-Tracing Just-In-Time Compilation for RPython*. PhD thesis, Heinrich Heine University Düsseldorf, 2014.

[23] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijałkowski, Michael Leuschel, Samuele Pedroni, and Armin Rigo. Allocation Removal by Partial Evaluation in a Tracing JIT. In *Proceedings of the 2011 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM*, 2011.

[24] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijałkowski, and Armin Rigo. Tracing the Meta-Level: PyPy's Tracing JIT Compiler. In *Proceedings of the 2009 Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems, ICOOOLPS*, 2009.

[25] Carl Friedrich Bolz and Armin Rigo. How to Not Write Virtual Machines for Dynamic Languages. In *Proceedings of the 2007 Workshop on Dynamic Languages and Applications, DYLA*, 2007.

[26] Per Bothner. Kawa: Compiling Scheme to Java. 1998.

[27] Preston Briggs, Keith D. Cooper, and Linda Torczon. Improvements to Graph Coloring Register Allocation. *ACM Transactions on Programming Languages and Systems, TOPLAS*, 16(3), 1994.

[28] Alejandro Caro. A Novel 64 Bit Data Representation for Garbage Collection and Synchronizing Memory. *Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology*, 1997.

[29] John Cavazos, J. Eliot B. Moss, and Michael F. P. O'Boyle. Hybrid Optimizations: Which Optimization Algorithm to Use? In *Proceedings of the 2006 International Conference on Compiler Construction, CC*, 2006.

[30] Gregory J. Chaitin. Register Allocation & Spilling via Graph Coloring. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, 1982.

[31] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register Allocation via Coloring. *Computer Languages*, 1981.

[32] Craig Chambers and David Ungar. Customization: Optimizing compiler Technology for SELF, a Dynamically-Typed Object-Oriented Programming Language. In *Proceedings of the 1989 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*, 1989.

[33] Craig Chambers, David M. Ungar, and Elgin Lee. An Efficient Implementation of SELF - a Dynamically-Typed Object-Oriented Language Based on Prototypes. In *Proceedings of the 1989 ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA*, 1989.

[34] Christopher Chedeau and Didier Verna. JSPP: Morphing C++ into JavaScript, 2012.

[35] Chris J. Cheney. A Nonrecursive List Compacting Algorithm. *Communications of the ACM*, 13(11), 1970.

[36] Maxime Chevalier-Boisvert. *On the Fly Type Specialization without Type Analysis*. PhD thesis, Université de Montréal, 2015.

[37] Maxime Chevalier-Boisvert and Marc Feeley. Simple and Effective Type Check Removal through Lazy Basic Block Versioning. In *29th European Conference on Object-Oriented Programming, ECOOP*, 2015.

[38] Maxime Chevalier-Boisvert and Marc Feeley. Interprocedural Type Specialization of JavaScript Programs without Type Analysis. In *30th European Conference on Object-Oriented Programming, ECOOP*, 2016.

[39] Fred C. Chow and John L. Hennessy. The Priority-Based Coloring Approach to Register Allocation. *ACM Transactions on Programming Languages and Systems, TOPLAS*, 12(4), 1990.

[40] William D. Clinger and Lars Thomas Hansen. Lambda, the Ultimate Label or a Simple Optimizing Compiler for Scheme. In *LISP and Functional Programming*, 1994.

[41] John Cocke and Jacob T. Schwartz. Programming Languages and Their Compilers: Preliminary Notes. 1970.

[42] Ole-Johan Dahl and Bjørn Myrhaug. Simula Implementation Guide. Publication S47. 1973.

[43] Daniele Cono D'Elia and Camil Demetrescu. Flexible On-Stack Replacement in LLVM. In *Proceedings of the International Symposium on Code Generation and Optimization, CGO*, 2016.

[44] L. Peter Deutsch and Allan M. Schiffman. Efficient Implementation of the Smalltalk-80 System. In *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages, POPL*, 1984.

[45] Charles Donnelly and Richard Stallman. Bison. The YACC-compatible Parser Generator.

[46] Karel Driesen, Urs Hölzle, and Jan Vitek. Message Dispatch on Pipelined Processors. In *9th European Conference on Object-Oriented Programming, ECOOP*, 1995.

[47] R. Kent Dybvig. The Development of Chez Scheme. In *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming, ICFP*, 2006.

[48] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.

[49] M. Anton Ertl, David Gregg, Andreas Krall, and Bernd Paysan. Vmgen - A generator of Efficient Virtual Machine Interpreters. *Software: Practice and Experience*, 32(3), 2002.

[50] Michael Fairhurst. Inverse Pointer Unboxing. 2017.

[51] Marc Feeley. Gambit Scheme Compiler v4.9.3, February 2019.

[52] Marc Feeley. Polling Efficiently on Stock Hardware. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture, FPCA*, 1993.

[53] Marc Feeley. SRFI-4: Homogeneous Numeric Vector Datatypes. *Scheme Request for Implementation, SRFI*, 1999.

[54] Marc Feeley. Speculative Inlining of Predefined Procedures in an R5RS Scheme to C Compiler. In *Implementation and Application of Functional Languages, IFL*, 2007.

[55] Matthew Flatt and PLT. Reference: Racket. Technical Report PLT-TR-2010-1, PLT Design Inc., 2010. `https://racket-lang.org/tr1/`.

[56] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. Trace-Based Just-In-Time Type Specialization for Dynamic Languages. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*, 2009.

[57] Andreas Gal, Christian W. Probst, and Michael Franz. HotpathVM: An Effective JIT Compiler for Resource-Constrained Devices. In *Proceedings of the 2nd International Conference on Virtual Execution Environments, VEE*, 2006.

[58] Matthew Gaudet. Experiments in Sharing Java VM Technology with CRuby. RubyKaigi, 2015.

[59] Matthew Gaudet and Mark Stoodley. Rebuilding an Airliner in Flight: A Retrospective on Refactoring IBM Testarossa Production Compiler for Eclipse OMR. In *Proceedings of the 8th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages, VMIL*, 2016.

[60] Nicolas Geoffray, Gaël Thomas, Julia L. Lawall, Gilles Muller, and Bertil Folliot. VMKit: a substrate for managed runtime environments. In *Proceedings of the 6th International Conference on Virtual Execution Environments, VEE*, 2010.

[61] Guillaume Germain. Concurrency Oriented Programming in Termite Scheme. In *Proceedings of the 2006 ACM SIGPLAN workshop on Erlang*, 2006.

[62] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, 1983.

[63] David Gudeman. Representing Type Information in Dynamically Typed Languages, 1993.

[64] Brian Hackett and Shu-yu Guo. Fast and Precise Hybrid Type Inference for JavaScript. In *Proceedings of the 2012 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*, 2012.

[65] Fritz Henglein. Global tagging optimization by type inference. *ACM SIGPLAN Lisp Pointers*, (1):205–215, 1992.

[66] Urs Hölzle, Craig Chambers, and David M. Ungar. Optimizing Dynamically-Typed Object-Oriented Languages with Polymorphic Inline Caches. In *5th European Conference on Object-Oriented Programming, ECOOP*, 1991.

[67] Urs Hölzle, Csraig Chambers, and David M. Ungar. Debugging Optimized Code with Dynamic Deoptimization. In *Proceedings of the 1992 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1992.

[68] Urs Hölzle and David M. Ungar. A Third-Generation SELF Implementation: Reconsiling Responsiveness with Performance. In *Proceedings of the 1994 ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA*, 1994.

[69] Urs Hölzle and David M. Ungar. Optimizing Dynamically-Dispatched Calls with Run-Time Type Feedback. In *Proceedings of the 1994 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*, 1994.

[70] Christian Humer, Christian Wimmer, Christian Wirth, Andreas Wöß, and Thomas Würthinger. A Domain-Specific Language for Building Self-Optimizing AST Interpreters. In *Proceedings of the 2014 International Conference on Generative Programming: Concepts and Experiences, GPCE*, 2014.

[71] Urs Hölzle. *Adaptive Optimization for SELF: Reconciling High Performance with Exploratory Programming*. Department of Computer Science, Stanford University, 1994.

[72] Intel. Intel® 64 and IA-32 Architectures Software Developer's Manual. 2016.

[73] Suresh Jagannathan and Andrew K. Wright. Effective Flow Analysis for Avoiding Run-Time Checks. In *Proceedings of the Second International Symposium on Static Analysis, SAS*, 1995.

[74] Suresh Jagannathan and Andrew K. Wright. Flow-directed Inlining. In *Proceedings of the 1996 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*, 1996.

[75] Erik Johansson and Konstantinos Sagonas. Linear Scan Register Allocation in a High-Performance Erlang Compiler. In *Proceedings of the 4th International Symposium on Practical Aspects of Declarative Languages, PADL*, 2002.

[76] Thomas Johnsson. Lambda Lifting: Transforming Programs to Recursive Equations. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture, FPCA*, 1985.

[77] Richard Jones, Antony Hosking, and Eliot Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. 2011.

[78] Madhukar N. Kedlaya, Behnam Robatmili, Calin Cascaval, and Ben Hardekopf. Deoptimization for Dynamic Language JITs on Typed, Stack-based Virtual Machines. In *Proceedings of the 10th International Conference on Virtual Execution Environments, VEE*, 2014.

[79] Andrew W. Keep and R. Kent Dybvig. A Nanopass Framework for Commercial Compiler Development. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming, ICFP*, 2013.

[80] Richard Kelsey, William D. Clinger, and Jonathan Rees. Revised^5 Report on the Algorithmic Language Scheme. 1998.

[81] David A. Kranz, Richard Kelsey, Jonathan Rees, Paul Hudak, and James Philbin. ORBIT: An Optimizing Compiler for scheme. In *Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction*, 1986.

[82] Monica Lam, Ravi Sethi, JD Ullman, and Alfred Aho. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 2006.

[83] Nurudeen Lameed and Laurie J. Hendren. A Modular Approach to On-Stack Replacement in LLVM. In *Proceedings of the 9th International Conference on Virtual Execution Environments, VEE*, 2013.

[84] Peter J Landin. The Mechanical Evaluation of Expressions. *The Computer Journal*, 6(4), 1964.

[85] Chris Lattner and Vikram S. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization, CGO*, 2004.

[86] Tim Lindholm, Frank Yellin, Gilad Bracha, Alex Buckley, and Daniel Smith. *The Java Virtual Machine Specification*. 2018.

[87] Francesco Logozzo and Herman Venter. RATA: Rapid Atomic Type Analysis by Abstract Interpretation - Application to JavaScript Optimization. In *Proceedings of the 2010 International Conference on Compiler Construction, CC*, 2010.

[88] Stefan Marr, Tobias Pape, and Wolfgang De Meuter. Are We There Yet?: Simple Language Implementation Techniques for the 21st Century. *IEEE Software*, 31(5), 2014.

[89] John McCarthy. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. *Communications of the ACM*, 3(4), 1960.

[90] John McCarthy. An Interesting LISP Function. *ACM LISP Bulletin*, (3), 1979.

[91] Richard McDougall and Jim Mauro. *Solaris Internals: Solaris 10 and OpenSolaris Kernel Architecture*. Pearson Education, 2006.

[92] Microsoft. Common Language Runtime (CLR).

[93] Jan Midtgaard. Control-Flow Analysis of Functional Programs. *ACM Computing Surveys, CSUR*, 44(3), 2012.

[94] Matthew Might. *Environment Analysis of Higher-Order Languages*. PhD thesis, Georgia Institute of Technology, 2007.

[95] Mike Pall. The LuaJIT Project. `http://luajit.org`.

[96] Hanspeter Mössenböck and Michael Pfeiffer. Linear Scan Register Allocation in the Context of SSA Form and Register Constraints. In *Proceedings of the 2002 International Conference on Compiler Construction, CC*, 2002.

[97] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[98] Open Source Initiative. The BSD 3-Clause License. `http://opensource.org/licenses/BSD-3-clause`.

[99] Guilherme Ottoni. HHVM JIT: a profile-guided, region-based compiler for PHP and Hack. In *Proceedings of the 2018 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*, 2018.

[100] Julien Pagès. *Une Machine Virtuelle en Héritage Multiple Basée sur le Hachage Parfait*. PhD thesis, Université de Montpellier, 2016.

[101] Julien Pagès. Twopy: A Just-In-Time Compiler For Python Based On Code Specialization. Virtual Machines and Intermediate Languages, VMIL, 2018.

[102] Michael Paleczny, Christopher Vick, and Cliff Click. The Java Hotspot Server Compiler. In *Proceedings of the 2001 Symposium on Java Virtual Machine Research and Technology Symposium*, volume 1, 2001.

[103] Terence J. Parr and Russell W. Quong. ANTLR: A Predicated-LL(k) Parser Generator. *Software: Practice and Experience*, 25(7), 1995.

[104] Vern Paxson. Flex. Fast Lexical Analyzer Generator.

[105] Simon L. Peyton Jones and Simon Marlow. Secrets of the Glasgow Haskell Compiler Inliner. *Journal of Functional Programming*, 12(5), 2002.

[106] Filip Pizlo. All About JavaScriptCore's Many Compilers. *Invited talk at SPLASH-I*, 2018.

[107] Massimiliano Poletto and Vivek Sarkar. Linear Scan Register Allocation. *ACM Transactions on Programming Languages and Systems, TOPLAS*, 21(5), 1999.

[108] Armin Rigo. Representation-Based Just-In-Time Specialization and the Psyco Prototype for Python. In *Proceedings of the 2004 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM*, 2004.

[109] Armin Rigo and Samuele Pedroni. PyPy's Approach to Virtual Machine Construction. In *Companion to the 2006 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA*, 2006.

[110] Baptiste Saleil. LC Scheme JIT Compiler. `https://github.com/bsaleil/lc`.

[111] Baptiste Saleil. Étude de kit de développement de compilateurs et machines virtuelles. Master's thesis, Université Montpellier 2, 2013.

[112] Baptiste Saleil and Marc Feeley. Code Versioning and Extremely Lazy Compilation of Scheme. In *Scheme and Functional Programming Workshop*, 2014.

[113] Baptiste Saleil and Marc Feeley. Type Check Removal Using Lazy Interprocedural Code Versioning. In *Scheme and Functional Programming Workshop*, 2015.

[114] Baptiste Saleil and Marc Feeley. Interprocedural Specialization of Higher-Order Dynamic Languages without Static Analysis. In *31st European Conference on Object-Oriented Programming, ECOOP*, 2017.

[115] Baptiste Saleil and Marc Feeley. Interprocedural Specialization of Higher-Order Dynamic Languages without Static Analysis (Artifact). In *31st European Conference on Object-Oriented Programming, ECOOP*, 2017.

[116] Baptiste Saleil and Marc Feeley. Building JIT Compilers for Dynamic Languages with Low Development Effort. In *Proceedings of the 10th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages, VMIL*, 2018.

[117] Dipanwita Sarkar, Oscar Waddell, and R. Kent Dybvig. A Nanopass Infrastructure for Compiler Education. In *Proceedings of the 9th ACM SIGPLAN International Conference on Functional Programming, ICFP*, 2004.

[118] Christopher Graham Seaton. *Specialising Dynamic Techniques for Implementing the Ruby Programming Language*. PhD thesis, University of Manchester, 2015.

[119] Bernard Paul Serpette and Manuel Serrano. Compiling Scheme to JVM Bytecode: A Performance Study. In *Proceedings of the 7th ACM SIGPLAN International Conference on Functional Programming, ICFP*, 2002.

[120] Manuel Serrano. Control Flow Analysis: A Functional Languages Compilation Paradigm. In *Proceedings of the 1995 ACM Symposium on Applied Computing, SAC*, 1995.

[121] Manuel Serrano and Pierre Weis. Bigloo: A Portable and Optimizing Compiler for Strict Functional Languages. In *Proceedings of the Second International Symposium on Static Analysis, SAS*, 1995.

[122] Zhong Shao and Andrew W. Appel. Space-Efficient Closure Representations. In *LISP and Functional Programming*, pages 150–161, 1994.

[123] Stanley T. Shebs. Implementing Primitive Datatypes for Higher-Level Languages. Technical Report UUCS-88-020, University of Utah, 1988.

[124] Alex Shinn, John Cowan, and Arthur A. Gleckler. Revised^7 Report on the Algorithmic Language Scheme. 2013.

[125] Olin Shivers. Control-Flow Analysis in Scheme. In *Proceedings of the 1988 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*, 1988.

[126] Olin Shivers. *Control-flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie Mellon University, 1991.

[127] Michael D. Smith, Norman Ramsey, and Glenn H. Holloway. A Generalized Algorithm for Graph-Coloring Register Allocation. In *Proceedings of the 2004 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*, 2004.

[128] Rodrigo Sol, Christophe Guillon, Fernando Magno Quintão Pereira, and Mariza A. S. Bigonha. Dynamic Elimination of Overflow Tests in a Trace Compiler. In *Proceedings of the 2011 International Conference on Compiler Construction, CC*, 2011.

[129] Michael Sperber, R Kent Dybvig, Matthew Flatt, Anton Van Straaten, Robby Findler, and Jacob Matthews. Revised^6 Report on the Algorithmic Language Scheme. 2007.

[130] Guy L Steele Jr. Rabbit: A compiler for Scheme. Technical report, Massachusetts Institute of Technology, 1978.

[131] P. Steenkiste. The Implementation of Tags and Run-Time Type Checking. In P. Lee, editor, *Topics in Advanced Language Implementation*, chapter 7, pages 157–188. 1991.

[132] Mark Stoodley. Eclipse OMR: A Modern Toolkit for Building Language Runtimes. EclipseCON, 2016.

[133] Gregory T Sullivan, Derek L Bruening, Iris Baron, Timothy Garnett, and Saman Amarasinghe. Dynamic Native Optimization of Interpreters. In *Proceedings of the 2003 Workshop on Interpreters, Virtual Machines and Emulators, IVME*, 2003.

[134] Gerald J. Sussman and Guy L. Steele, Jr. An Interpreter for Extended Lambda Calculus. Technical report, 1975.

[135] Cisco Systems. Chez Scheme Compiler v9.5.3, March 2019.

[136] Omri Traub, Glenn H. Holloway, and Michael D. Smith. Quality and Speed in Linear-scan Register Allocation. In *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*, 1998.

[137] David M. Ungar and Randall B. Smith. Self: The Power of Simplicity. In *Proceedings of the 1987 ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA*, 1987.

[138] Christian Wimmer and Stefan Brunthaler. ZipPy on Truffle: A Fast and Simple Implementation of Python. In *Proceedings of the 2013 companion publication for Conference on Systems, Programming, and Applications: Software for Humanity, SPLASH*, 2013.

[139] Christian Wimmer and Michael Franz. Linear scan register allocation on SSA form. In *Proceedings of the International Symposium on Code Generation and Optimization, CGO*, 2010.

[140] Christian Wimmer and Hanspeter Mössenböck. Optimized interval splitting in a linear scan register allocator. In *Proceedings of the 1st International Conference on Virtual Execution Environments, VEE*, 2005.

[141] Christian Wimmer and Thomas Würthinger. Truffle: A Self-Optimizing Runtime System. In *Proceedings of the 2012 Conference on Systems, Programming, and Applications: Software for Humanity, SPLASH*, 2012.

[142] Wengfai F. Wong. Optimizing floating point operations in scheme. *Computer Languages*, 25(2), 1999.

[143] Andreas Wöß, Christian Wirth, Daniele Bonetta, Chris Seaton, Christian Humer, and Hanspeter Mössenböck. An Object Storage Model for the Truffle Language Implementation Framework. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages and Tools, PPPJ*, 2014.

[144] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. One VM to Rule Them All. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward!*, 2013.

[145] Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. Self-Optimizing AST Interpreters. In *Proceedings of the 8th symposium on Dynamic languages, DLS*, 2012.

[146] Ming-Ho Yee. CS 7600 Survey Paper: On-Stack Replacement. 2018.